UiO : Department of Informatics
University of Oslo

Energy Consumption Investigations in WSN using OMNeT++

Espen Nilsen, master thesis spring 2013

Energy Consumption Investigations in WSN using OMNeT++

Espen Nilsen

15th May 2013

Abstract

Wireless sensor networks have attracted a lot of interest by various industries for its ability to perform real time measurements in a cost effective way. Since the sensor nodes often rely on batteries to perform their task, a lot of research have been focused on minimizing the energy consumption in WSN.

In this project TDMA based WSN was investigated to see if the energy consumption could be minimized by manipulating the nodes communication range. Though a lot of research has been focused on similar problems exist these are mostly focused on non TDMA network.

The problem was partially analysed analytically, but the mainly assessed by simulation, using the OMNeT++ simulation framework. Since OMNeT++ is not a simulator by it self, the MiXiM model framework that focuses on the simulation of wireless networks was used.

It was shown that a minimum point for energy consumption can be found. However, a small degradation in the quality of the links may cause a dramatic increase in the energy consumption due to retransmissions.

Contents

List of Figures

List of Tables

Preface

This thesis concludes my Master degree at the Department of Informatics at the University of Oslo. I wish to thank my supervisors; Professor Knut Øvsthus at Bergen University College, and Professor Pål Orten at the University of Oslo and ABB Corporate Research.

Lastly thanks to all my friends and family and specially my girlfriend Sofie, for all the support.

Chapter 1

Introduction

1.1 Background and Motivation

A wireless sensor network (WSN) is a collection of sensors that measures or monitors physical conditions in an area of interest, and gathers measurement data in a central unit by means of wireless communication. Advances in the fields of sensing, computing and communication have led to the development of inexpensive low-powered sensor nodes allowing the deployment of WSNs for various applications. As with many technologies, defence applications have been a driver for early research and development [7]. Today however, drivers include numerous applications both commercial an research oriented. Self configurability and the fact that no wiring is needed, make WSN a cost saving and flexible alternative to old wired sensor systems, already deployed by several industries. With a high number of sensors, the planning and installation of wired networks is a complex and time consuming operation, and the cost of manpower, cables and conduit makes it an accordingly expensive solution. In the process industry, WSN has been shown to reduce the installation cost compared to wired sensor systems by as much as 90% [18].

In many application scenarios, fixed power supplies will not be available, and the nodes have to rely on batteries to perform their tasks. In a WSN, each node has the role of both data originator and data router. The malfunctioning of a few nodes may lead to significant changes in the network topology and might require rerouting of packets and reorganisation of the network [5]. Additionally, some applications might require long lifetimes (years) for both the network and individual nodes. Hence, a lot of research has been aimed at minimising the energy consumption in WSN. Although a sensor node consist of units such as one or more sensors, the controller and the memory, most energy is consumed by the radio [10]. Thus, various protocols have been proposed to minimise the energy consumption of the radio. These may be categorized as topology control protocols and sleep management protocols [15]. Topology control protocols manipulate the communication ranges of nodes to manage the number of nodes to be considered as neighbours. In [13] 12 reasons to avoid routing over many short hops are discussed. These reasons include; energy

consumption, end-to-end reliability and delay. In [15] it was shown that the energy consumption by receiving nodes is the main contributor to total energy consumption in WSN and that the energy optimal transmission range is short. However, in a TDMA based network, non communication nodes can be put to sleep.

1.2 Scope

The purpose of this thesis is to investigate how energy consumption in TDMA based WSN can be minimised by manipulating the node range. In particular, random network topologies are considered to a assess whether or not it is beneficial to route over longer hops by accepting a poorer link quality. Only fixed networks are considered, and it is assumed that all communicating nodes are assigned non overlapping time slots. Focus has also been laid on describing the OMNeT++ simulation frame work and the MiXiM model framework used for simulation.

1.3 Thesis Structure

This document is organised as follows:

- Chapter 1 (this chapter) gives the background/motivation and purpose of the thesis.
- Chapter 2 provides the necessary theory needed to understand the work presented in this document.
- Chapter 3 presents the OMNeT++ simulation framework and the MiXiM model framework. How the simulation models was developed using this framework is also described in this chapter, and various other tools are presented.
- Chapter 4 describes the performed simulation scenarios and presents the results.
- Chapter 5 draws conclusion based on the results and experiences made while working on this project.

Chapter 2

Theory

This chapter presents the theory needed to understand the work presented in this document.

2.1 IEEE 802.15.4

The IEEE 802.15.4 standard focuses on short range communication and is characterized by low-data-rates, long battery life times (e.g months or years), very low complexity and low hardware costs [27]. The standard specifies the physical (PHY) and medium access layer (MAC) that can be used to design low-powered radios for wireless sensor nodes. Several PHYs are specified by the standard with different data-rates and modulation techniques. This project focuses on the *direct sequence spread spectrum* (DSSS) PHY, employing offset quadrature phase-shift keying (O-QPSK) , and operating in the 2.4 GHz frequency band (see subsection 2.1.3). Additionally, only nonbeacon enabled peer-to-peer networks are considered.

2.1.1 MAC Protocol

The IEEE 802.15.4 standard offers two types of channel access mechanism; slotted and unslotted CSMA-CA. For a nonbeacon enabled peer-to-peer network the unslotted version is used. Whenever a device wishes to transmit data frames it waits for a random number of time periods (backoff). If the channel is found to be idle, the device transmits its data immediately after the random backoff. If the channel is busy, the device waits an additional backoff time before trying to access the channel again. Acknowledgements, if used, are sent without using the CSMA-CA mechanism.

A flow chart of the unslotted CSMA-CA algorithm is shown in figure 2.2. For each transmission a device keeps track of the following variables; NB and BE. NB is the number of times the CSMA-CA algorithm has been required to back off for the current transmission; and BE is the current backoff exponent which relates to the number of backoff periods the device must wait before attempting to access the channel. Initially BE is set to

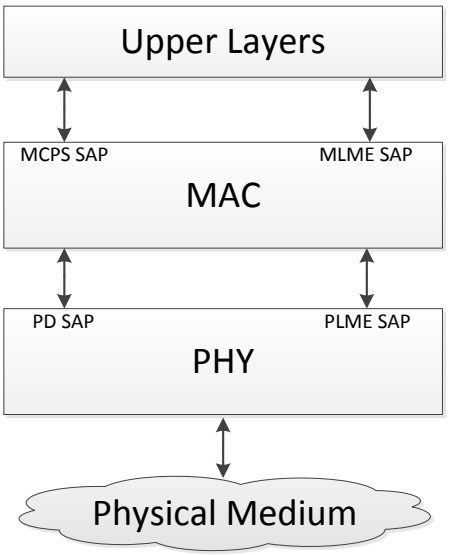

Figure 2.1: IEEE 802.15.4 Device Architecture [20]

a minimum value *aMacMinBE*. For every consecutive time a device must back off because of a busy channel, BE is incremented until the transmission succeeds, or a maximum value, *macMaxBE*, is reached. If the channel is continuously busy, a device will perform a maximum number of backoffs as defined by a constant, macMaxCSMABackoffs, before discarding the frame.

2.1.2 Receiver Sensitivity Definition

IEEE 802.15.4 defines the receiver sensitivity as *"The Lowest input power for which the PER conditions are met"*, where PER is the *Packet Error Rate* defined as the average fraction of transmitted packets that are not correctly received. When this definition of sensitivity is used, it is assumed that only thermal noise is present. For a device to be compliant with the standard, a sensitivity of -85 dBm or better is required.

2.1.3 O-QPSK PHY

This section briefly introduces the O-QPSK concepts considered in this document.

Bit Error Performance

The bit error of the IEEE 802.15.4 O-QPSK scheme has a somewhat better performance over regular QPSK. The O-QPSK bit error probability is given by the standard as [20]:

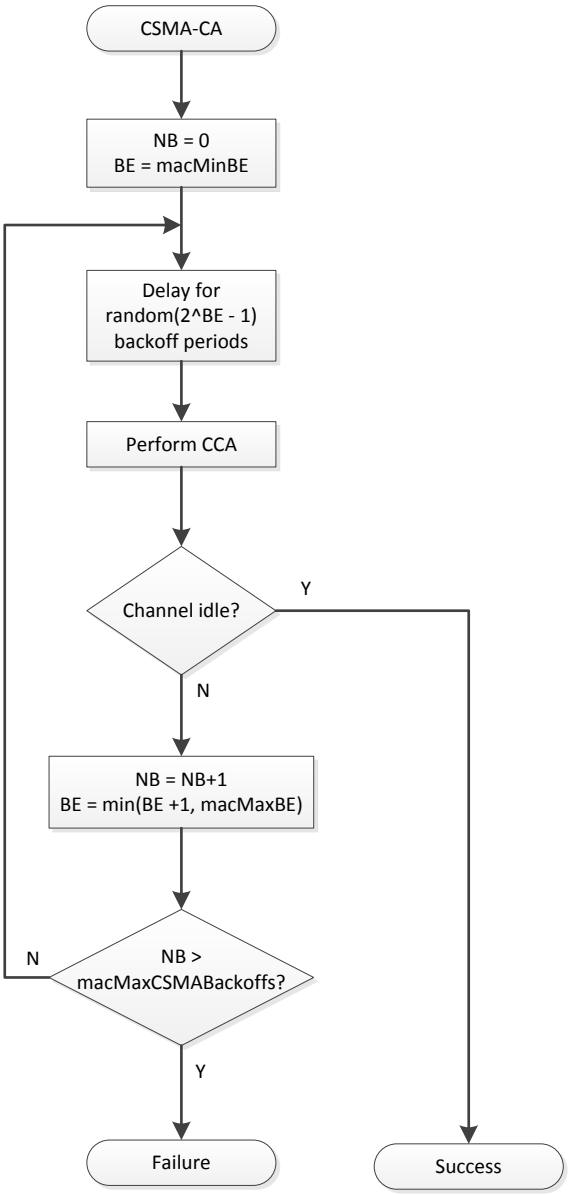

Figure 2.2: Unslotted version of the CSMA-CA algorithm [20]

$$P_b = \frac{8}{15} \cdot \frac{1}{16} \sum_{k=2}^{16} e^{20 \cdot SINR \cdot (\frac{1}{k} - 1)} \tag{2.1}$$

Frame Format

The structure of the *PHY protocol data unit* is depicted in figure 2.3. The synchronization header (SHR), which is transmitted first, consist of *preamble* and a *start frame delimiter* (SFD) indicating the end of the synchronization header. The *PHY header* (PHR) consist of 8 bits, where the firs 7 indicates the size of the frame, and the last bit is reserved. The last field in the PPDU is the PHY payload, or *PHY service data unit* (PSDU), which is the MAC frame handed to the PHY for transmission. The PPDU field lengths are summarised in table 2.1.

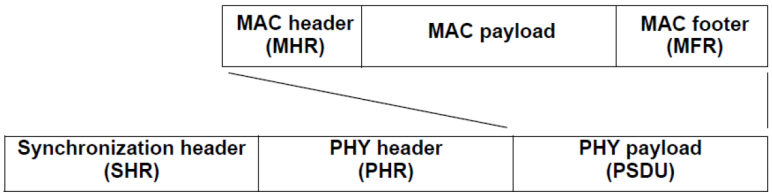

Figure 2.3: IEEE 802.15.4 O-QPSK PPDU format [20]

PPDU field	Length (bytes)
SHR	5
PHR	1
PSDU (DATA)	9-127
PSDU (ACK)	5

Table 2.1: PPDU field lengths

2.2 Transceivers

For communication, both a transmitter and a receiver is required in a sensor node. The transmitter translates digital information into waveforms that propagate the wireless channel, while the receiver interprets these waveforms to reproduce the information that was sent. Both these task are often grouped into a single device called a *transceiver*. Several vendors produce single chip solutions that are compliant with the IEEE 802.15.4 standard (e.g. Freescale, Atmel, Ember). However, this project focuses on the Texas Instruments/Chipcon CC2420 transceiver.

2.2.1 The Chipcon CC2420 Transceiver

A popular choice of transceiver for WSN is the Texas Instruments/Chipcon CC2420 transceiver. It implements the O-QPSK PHY layer as described

Mode	Value	Unit
Power Down mode (PD)	0.02	µA
Idle mode (IDLE)	20	µA
Receive Mode	18.8	mA

Table 2.2: Current consumption in different modes of operation at 3.3 V

Output Power[dBm]	Current Consumption [mA]
0	17.4
-1	16.5
-3	15.2
-5	13.9
-7	12.5
-10	11.2
-15	9.9
-25	8.5

Table 2.3: Typical current consumption for the CC2420 measured at 3.3 V for different output power configurations [6]

by the IEEE 802.15.4 standard with the required support for the standards MAC protocol [6].

Why this transceiver was chosen

Although there are other available IEEE 802.15.4 compliant transceivers used for WSN, the CC2420 chip was chosen because of its widespread use in the research community. According to [23], the CC2420 transceiver is the most widely used chip in sensor network research. Additionally, several simulation frameworks offer models for this particular radio chip [24].

Power Consumption

The current consumption in different modes of operation measured at 3.3 v is shown in table

2.3 Radio Frequency Propagation

In this section the basics of radio frequency propagation is discussed.

2.3.1 Free Space

A starting point for modelling the characteristics of a wireless channel, is to assume that the signal attenuates as if propagating through ideal free space. This implies that the region between the sender and receiver is totally free of objects that might absorb, scatter, or otherwise influence the signal in its path. It also assumes no reflections, and that the wireless medium itself does not absorb any signal energy. In free space, the transmit power

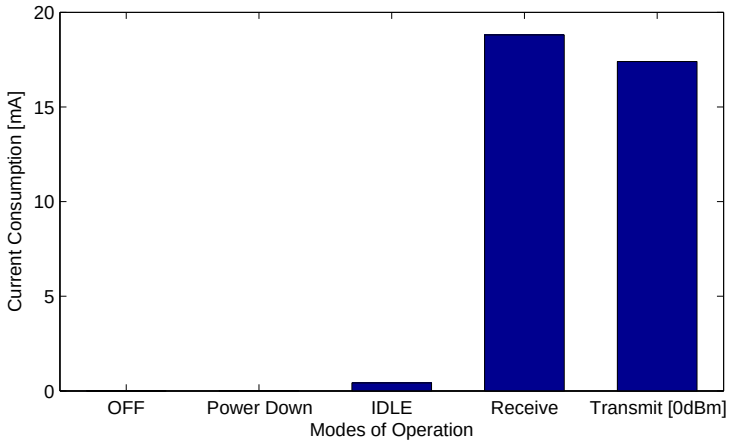

Figure 2.4: CC2420 Current consumption in the different modes of operation

radiating from an ideal isotropic antenna, is attenuated by a factor $L_s(d)$, referred to as *free space loss* [22].

$$L_s(d) = \left(\frac{4\pi d}{\lambda}\right)^2 \tag{2.2}$$

where;

- d is the distance from the transmitter
- λ is the wavelength of the transmitted signal

The assumption of an isotropic antenna, means that the transmitted signal energy is considered as radiating spherically, and uniformly, from a point source (also called an *isotropic radiator*[22]).

2.3.2 Mean Path Loss

In environments other than free space it is often assumed that the mean path loss $\overline{L_s}(d)$ has a proportional relationship with the distance as follows [22]:

$$\overline{L_s}(d) \propto \left(\frac{d}{d_0}\right)^{\alpha} \tag{2.3}$$

where α is the *path loss exponent* and d_0 is a reference distance to a point in the *far-field* of the transmitting antenna

- α is the *path loss exponent*
- d_0 is a reference distance to a point in the far field of the antenna

In short range systems like IEEE 802.15.4, d_0 is in the range of 1 m [16]. Values for the path loss exponent, which is dependent on the environment, normally ranges from 2 (free space) to 6 (heavily obstructed and shadowed areas) [28]. In [11] the authors refers to measurement carried out in a factory environment where α was found to be in the range of 2-3. A typical value for α in both indoor and outdoor environments is referred to in [28] as 3.

In this thesis the effects of the near field communication is not considered and the received signal power from a source transmitting with an output power Ptx is assumed to be given by:

$$P_r = \frac{P_{tx} \cdot \lambda^2}{(4\pi)^2 \cdot d^\alpha} \tag{2.4}$$

2.3.3 Link Quality Indication

The Link Quality Indication (LQI) measurement is a characterization of the strength and/or quality of a received packet [20]. LQI may be used by routing protocols as metric, to avoid using poor quality links that result in high packet loss probabilities. Energy detection, signal-to-noise ratio estimations, or a combination of these methods are suggested by the IEEE 802.15.4 to be used in the measurement of the LQI. The standard does not however, specify how these measurements shall be implemented, and as a result, different hardware vendors provide different solutions.

The CC2420 radio chip provides a received signal strength indicator (RSSI) and an average correlation value that can be used in calculating the LQI. In the data sheet the, LQI is recommended to be found empirically through PER measurements in conjunction with the correlation value[6]. RSSI could be used directly, but has the disadvantage that interference imposed on the signal may increase the measured LQI, when the true link quality is actually reduced.

In [25], twenty-four hour measurements, using CC2420 rado chips, were carried out to assess the quality of links in a factory environment. Both RSSI and the correlation value where investigated separately for comparison. It was shown that the correlation value alone provided a higher statistical relation with the PER. However, it is also pointed out that the for a high PER the correlation value will overestimate the LQI, since the lost packets are not considered.

2.3.4 Packet Error Rate

Packet error rates can be calculated by:

$$\text{PER} = 1 - (1 - P_b)^{n_{bits}}, \tag{2.5}$$

where;

- P_b is the probability of a bit error
- $n^{n_{bits}}$ is the packet size in bits

2.4 Energy Consumption

In this document only energy consumption in transmit and receive mode are considered. It is assumed that switching time between modes are low so that the energy consumed by transmitting and receiving wil be dominant. The energy consumption will be assessed by counting packets.

2.4.1 Transmit mode

There are 8 output power configurations available in the CC2420 transceiver, each associated with a corresponding current consumption depending on the supply voltage. Letting I_i represent the current consumption in configuration i, where $i \in \{0, -1, -3, -5, -7, -10, -15, -25\}$ dBm, the power consumed in a given output power configuration can be expressed as:

$$P_i = U \cdot I_i, \tag{2.6}$$

where:

- P_i is measured in Watts.
- U is the supply voltage in volts.

The energy consumed by transmissions is then:

$$E_{tx} = P_i \cdot T_{tx} \tag{2.7}$$

where

- E_{tx} is measured in joules.
- T_{tx} is the total time spent in transmit mode expressed in seconds.

The total amount of time spent in transmit mode is dependent on both the number of packets sent, and the time it takes to transmit each packet. Only considering data packets and acknowledgements, T_{tx} can be calculated by:

$$T_{tx} = N_{txData} \cdot \frac{L_{Data}}{R} + N_{txAck} \cdot \frac{L_{Ack}}{R} \tag{2.8}$$

where:

- N_{txData} and N_{txAck} are the number of transmitted data packets and acknowledgements respectively
- L_{Data} and L_{Ack} are the packet sizes of data packets and acknowledgements respectively measured in bits.
- R is the data rate in bps.

Parameter	Value	Unit
Data Packet Length	26	bytes
Ack Packet Length	11	bytes
Supply Voltage	3.3	volts
Current Consumption in receive mode	18.8	mA
Current consumption in transmit mode	variable (see table 2.3)	mA

Table 2.4: Parameters used in the calculation of energy consumption

2.4.2 Receive Mode

Following the notation introduced in the previous subsection, the energy consumed in receive mode (denoted E_{rx}) is given by:

$$E_{rx} = P_i \cdot T_{rx} \tag{2.9}$$

Where the time spent in receive mode, T_{rx}, is calculated as:

$$T_{rx} = N_{rxData} \cdot \frac{L_{Data}}{R} + N_{rxAck} \cdot \frac{L_{Ack}}{R} \tag{2.10}$$

- N_{rxData} and N_{rxAck} are the number of received data packets and acknowledgements respectively
- L_{Data} and L_{Ack} are the packet sizes of data packets and acknowledgements respectively measured in bits.
- R is the data rate in bps.

2.5 Analytical Investigation

In order to investigate the problem analytically, the extreme case of a network containing an infinite number of nodes is considered. The density of nodes in such a network, denoted λ, is consequently also infinite. Routing in a network of infinite node density is of course impossible. For the sake of simplicity, it is assumed that all nodes have a method for deciding how packets should be forwarded, and each packet is relayed to the farthest possible node in that direction limited by its range. The number of hops needed to transfer a packet to a sink located at a distance D from the sender is then given by:

$$N = \lceil \frac{D}{d} \rceil, \tag{2.11}$$

where:

- $\lceil x \rceil$ denotes the ceiling function of x
- d is the node range

- D is the total distance between the sender and sink

The path from a sending node, A, to the sink follows a line topology as shown in figure 2.5. Each hop i, where $i \in \{1..(N-1)\}$, is of the same length d, according to the range of the nodes. Here it is assumed that all nodes have the same transmission power P_{tx}, and that the path loss only depends on the distance. The length of the N^{th} hop will of course vary depending on both the node range, and the total distance between the sink and node A, but for now, consider all N hops as being of length d. Furthermore, it is assumed that only one node is transmitting at a time (the nodes do not interfere with each other), and that no external sources of interference are present.

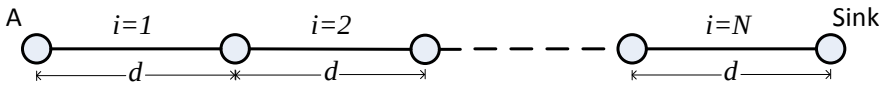

Figure 2.5: Line Topology

2.5.1 Expected number of transmissions

Continuing the scenario introduced in the previous section; let X_i be the random variable counting the number of transmissions on the i^{th} hop per packet sent from node A towards the sink. For simplicity, assume that the bit error rate associated with each hop is equal to P_b, so that:

$$P_{bi} = P_b, \forall i \in \{1..N\}, \tag{2.12}$$

and that the probabilities of individual bit errors are independent.

A transmission is regarded as successful if an acknowledgement (ACK), confirming the reception of the packet, is received by the sender. If the initial transmission of a packet fails (no ACK is received), the node will try to retransmit. The nodes have a parameter, denoted m, that limits the maximum number of transmission trials. Nodes that have tried to transmit a packet m times, discards the packet. When acknowledgements are used, there are two possible events resulting in a retransmission:

1. The transmitted data packet is lost on its way to the receiver.
2. The packet is successfully transferred to the receiver, but the ACK fails.

Thus, a successful transmission requires that all the bits in both the data and ACK packet must be transferred without error. Let q denote the packet failure rate (PFR) defined as the probability that a single transmission fails. Under the above assumptions:

$$q = 1 - (1 - P_b)^{L_{DATA}+L_{ACK}}, \tag{2.13}$$

where;

- L_{DATA} is the length of the data packet in bits
- L_{ACK} is the length of the ACK packet in bits

Expected Number of Transmissions on the First Hop

The expected value of X_1 (number of transmissions on the first hop) is given by:

$$E\{X_1\} = 1 \cdot q^0(1-q) + 2 \cdot q^1(1-q) + 3 \cdot q^2(1-q) + ... \tag{2.14}$$

$$... + m \cdot q^{m-1} \cdot (1-q) + m \cdot q^m \tag{2.15}$$

where:

- m is the maximum number of transmissions
- q is the probability of an unsuccessful transmission, and conversely, $(1-q)$ is the probability of a successful transmission

By recognising equation 2.14 as a geometric series, the expected number of transmissions can be expressed as (see appendix A.1.1):

$$E\{X_1\} = \frac{1-q^m}{1-q} \tag{2.16}$$

Equation 2.16 can be assessed by investigating whether correct results are given for the two extreme situations: $q = 0$ and $q = 1$. The limit of 2.16 as $q \to 0$ is given by:

$$\lim_{q \to 0} \frac{1-q^m}{1-q} = 1, \tag{2.17}$$

and by using *L'Hopital's rule*, the limt as $q \to 1$ is:

$$\lim_{q \to 1} \frac{1-q^m}{1-q} = \lim_{q \to 1} \frac{m \cdot q^{m-1}}{1} = m \tag{2.18}$$

When q is '0', no packet failures are expected, and a packet should be successfully transferred at first attempt. Conversely, when q is '1', all m transmissions should fail. This shows that, at least in the limit, equation 2.16 holds.

Multiple Hops

In the N hop scenario, equation 2.16 gives the expected number of transmissions in the first hop. On the i^{th} hop, the expected value depends on the probability that a packet has successfully traversed the (i-1) previous hops, $P(S_{i-1}) = (1-q^m)^{i-1}$, where $P(S_i)$ is the probability of a successful transmission on hop i. The expected number of transmissions on the i^{th} hop is then given by:

$$E\{X_i\} = E\{X_1\} \cdot P(S_{i-1}) = E\{X_1\} \cdot (1 - q^m)^{i-1} \tag{2.19}$$

The total number of expected transmissions over N hops is then:

$$\sum_{i=1}^{N} E\{X_i\} = E\{X_1\} \cdot \sum_{i=1}^{N} (1 - q^m)^{i-1} \tag{2.20}$$

In the limit as $q \to 1$ of equation 2.20, the total number of transmissions is expected to be m, because the packet can never go past the first hop. When $q \to 0$ however, only one transmission per hop is needed to reach the sink, and the expected number of transmission will then be equal to N. The limits of equation 2.20 are found in the following:

By the multiplication law of limits (when the limits exist):

$$\lim_{q \to c} \left(E\{X_i\} \cdot \sum_{i=1}^{N} (1 - q^m)^{i-1} \right) = \lim_{q \to c} E\{X_i\} \cdot \lim_{q \to c} \sum_{i=1}^{N} (1 - q^m)^{i-1}, \tag{2.21}$$

where;

- c is a constant

Because $\lim_{q \to c} E\{X_i\}$ is known for both $c = 0$ and $c = 1$, only $\lim_{q \to c} \sum_{i=1}^{N} (1 - q^m)^{i-1}$ needs to be resolved. By the power law of limits (when the limits exist):

$$\lim_{q \to 1} (1 - q^m)^{i-1} = \begin{cases} 1, & \text{if } i = 1 \\ 0, & \text{otherwise} \end{cases} \tag{2.22}$$

and

$$\lim_{q \to 0} (1 - q^m)^{i-1} = 1, \text{for all } i \tag{2.23}$$

So that:

$$\lim_{q \to 0} \sum_{i=1}^{N} (1 - q^m)^{i-1} = N \tag{2.24}$$

and:

$$\lim_{q \to 1} \sum_{i=1}^{N} (1 - q^m)^{i-1} = 1 \tag{2.25}$$

The limits of $E\{X_1\}$ as $q \to 0$ and $q \to 1$ i given by equations 2.17 and 2.18 respectively. Summing up; the limits of equation 2.20 are:

$$\lim_{q \to 0} \sum_{i=1}^{N} E\{X_i\} = 1 \cdot N = N \tag{2.26}$$

$$\lim_{q \to 1} \sum_{i=1}^{N} E\{X_i\} = m \cdot 1 = m \cdot 1 = m \tag{2.27}$$

Using equation 2.1 to calculate as formula for bit error

Chapter 3

Simulation Concepts and Tools

In computer simulations, a system or process is "imitated" over time [27]. A simplified model is made to reflect the parts of a real world system or process of interest, and all unnecessary details are left out. The goal is to, by conducting experiments on the model in the form of a computer program, gain a deeper understanding of the real world system or process.

3.1 Discrete Event Simulation

When modelling system behaviour by means of a discrete event simulator, the system is viewed as a discrete event system. The behaviour of such a system is governed by events occurring at discrete points in time, that result in distinct changes of the state of the system [9]. This means that everything taking place in the time interval between these discrete points are viewed as having negligible effect on the system state, and is not considered. The terminology used when describing discrete event simulation may vary somewhat in the literature. This section adopts the terminology used in [27].

Communications systems are frequently simulated using discrete event simulation, where events may represent packet transmission/reception, the expiring of a "hello-interval" timer etc. The idea is to jump from one event to the next, where each event may change the state of the system and also produce future events. All the events are managed with a suitable data structure referred to as FES (*future event set*) or FEL (*future event list*). The events are chronologically arranged and processed one at a time depending on the event type. According to [27], the following components are shared by all discrete-event simulators (list adopted from [27]):

- System state: a set of variables that describe the state of the system
- Clock: gives the current time during the simulation
- Future Event List: a data structure appropriate to manage the events
- Statistical Counters: a set of variables that contain statistical information about the performance of the system

- Initialisation routine: a routine that initialises the simulation model and sets the clock to 0
- Timing Routine: a routine that retrieves the next event from the future event list and advances the clock to the occurrence time of the event
- Event Routine: a routine that is called when a particular event occurs during simulation

Figure 3.1 shows the flow diagram of the time-advance algorithm, at the core of a discrete event simulator. At the start of the simulation the FEL is empty. Thus, the most important task of the initialisation routine is to supply the initial events. When all the events in the FEL have been processed, or some predefined condition is met, the simulation terminates and outputs statistical information.

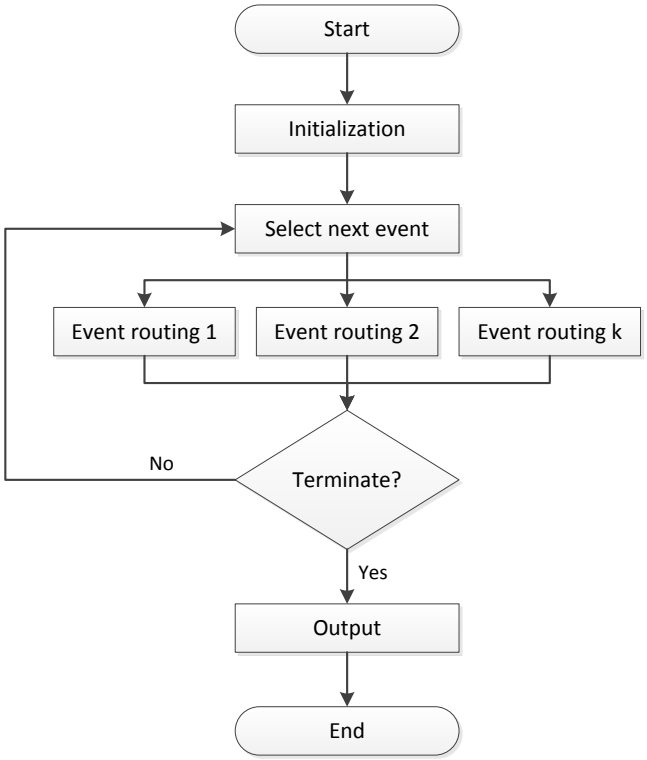

Figure 3.1: Flow diagram of the event-scheduling time-advance algorithm [27]

3.2 OMNeT++

OMNeT++, developed by Andras Varga at the Technical University of Budapest [26], is a modular, component-based C++ simulation library and

framework, primarily aimed at building network simulations. However, it was designed to be as general as possible, and as a result can be successfully used in other areas like the simulation of complex IT systems, queuing networks, or hardware architectures as well [1]. It provides the necessary tools and infrastructure to build simulation programs, but is not a simulator of anything concrete. Domain-specific functionality such as support for wireless sensor networks or internet protocols is provided by *model frameworks* as independent projects. OMNeT++ offers an Eclipse-based IDE and includes both a graphical user interface for interactive and animated runs, and a command-line interface for batch runs. The OMNeT++ simulation framework follows the principals of discrete event simulation.

3.2.1 Model Frameworks

Model frameworks offer component libraries and additional support for simulation in different application areas. Some examples of popular OMNeT++ network simulators are:

- INET/IMANET[2]
- Castalia[3]
- MiXiM[4]

3.2.2 Modelling Concepts

Modelling in OMNeT++ follows an hierarchical component-based architecture. Models are assembled from reusable components, called *modules*, that communicate with each other by exchanging messages. The active components, termed *simple modules*, are programmed using the C++ simulation library. Simple modules are at the lowest level in the module hierarchy and can be grouped into *compound modules*. The compound modules themselves may consist of other compound modules and the depth of module nesting is unlimited. Figure 3.2 gives an overview of this concept. The network module, or simply *network*, represents the top level in the module hierarchy and is also a complete OMNeT++ simulation model. The passing of messages between modules represents the events. Messages can be sent either directly to modules, or via predefined *gates* and *connections* (represented by the arrows in figure 3.2). Gates are the input and output interfaces of modules. If not sent directly, all messages leave the sending module on an output gate and enter the receiver module on an input gate. The linking of an output gate and an input gate is referred to as a connection. Connections may not span hierarchy levels as it would hinder model reuse.

3.2.3 NED

The structure of the OMNeT++ model, including module definitions (both simple and compound), gates and connections, is described with

OMNeT++'s own programming language NED (Network Description). The language supports familiar programming concepts like inheritance and interfaces, and has a Java-like packet structure to reduce the risk of name clashes between different modules [19]. In addition, modules described by the NED language may have *parameters*.

Network module

Modules are instances of module types defined in NED files. As mentioned in section 3.2.2, simple modules are at the lowest level in the module hierarchy, whereas the network module is at the highest. A network simulation model typically constitutes three levels of hierarchy; the network itself, the nodes, and the different protocol layers that each node consists of.

The type definition of a network consisting of three nodes could be written as:

```
network MyNetwork
{
    submodules:
        node1: Node;
        node2: Node;
        node3: Node;
    connections:
        node1.nodeGate <--> node2.nodeGate;
        node2.nodeGate <--> node3.nodeGate;
}
```

In the `submodules` section each sub-module is defined by `Node`, where `Node` is a simple or compound module with its own type definition. The nodes are connected in the `connections` section, where the double arrows mean that the connections are bidirectional. The nodes from a line topology with `node2` as the centre node. Using individual entries to declare each node in the network is fine when dealing with a small number of nodes. However, for larger networks the nodes should be declared as a vector of nodes. This can be done by replacing the individual node entries with `node[]`.

Compound Module Definition

With a protocol stack consisting of application, network, mac and physical layers, the implementation of `Node` might look like:

```
module Node
{
    parameters:
        int nodeAddress;
    gates:
        inout nodeGate;
    submodules:
```

```
        app: TrafficGenerator;
        netwl: RoutingProtocol;
        mac: MacLayer;
        phy: PhyLayer;
    connections:
        app.lowerLayer <--> netwl.higherLayer;
        netwl.lowerLayer <--> mac.higherLayer;
        mac.lowerLayer <--> phy.higherLayer;
        phy.lowerLayer <--> nodeGate;
}
```

As with the network module, `Node` is a compound module with sub-modules of its own. The gate used to connect the nodes in the network definition is defined in the `gates` section. Note that, in the `connections` section, the `PhyLayer` sub-module has to be connected to the gate of its parent module to allow communication with other nodes. In the `parameters` section, a parameter representing the address of the node is defined. Although only simple modules are implemented in C++, any parameter in the module hierarchy can be parsed and made use of by the programmer.

Simple Module Definitions

The layers in a protocol stack are typically implemented as simple modules. As an example, the implementation of `MacLayer` might look like:

```
simple MacLayer
{
    parameters:
        @class(MacLayer);
        int headerLength;
    gates:
        inout higherLayer;
        inout lowerLayer;
}
```

All simple module types must be declared using the `simple` keyword. As mentioned earlier in section 3.2.2, simple modules are programmed in C++ using the simulation library. The NED definitions provide the infrastructure of the model, and simple model type definitions must be linked to a corresponding C++ class. In the `parameters` section, this is done with the `@class(MacLayer)` NED *property* (NED poperties will be further discussed in the following section). Strictly speaking, it is not always necessary to specify the C++ class like this, since OMNeT++ chooses the class with the same name by default.

Parameters and Properties

Parameters are variables that belong to a module [19]. As described earlier in this section, they are declared in the NED files, but can be assigned values in configuration files as well. The configuration files are referred

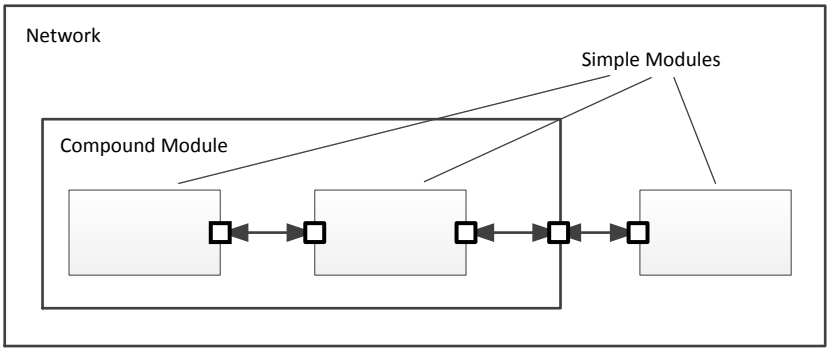

Figure 3.2: Simple and Compound modules

to as INI (initialization) files and is more thoroughly discussed in section 3.2.6. The different parameter types are; `double`, `int`, `bool`, `string` and `xml`. Additionally, NED *properties*, can be used to add extra information to parameters and almost all types of NED elements.

There are several ways of declaring parameters in NED, and its important to be aware of the effects. In general it is recommended that most parameters are assigned values in the INI files. Declaring a parameter in the following way would hard-code the assigned value in the model:

```
int headerLength = 20;
```

If the goal is to infer that `headerLength` should be '20', a safer way of declaring it would be:

```
int headerLenght = default(20);
```

With the above statement, `headerLength` is still assigned the value '20' by default, but is now allowed to be changed in the INI files. Leaving the parameter unassigned implies that the value **must** be set in the INI files, which is sometimes a good way of reminding the programmer to assign it an appropriate value (OMNeT++ will not start the simulation until all parameter are assigned values).

NED properties, denoted by the prefix `@`, are meta-data annotations used to add additional information to NED elements. The most important application for this, is to provide the simulation kernel with information about which C++ classes should be linked to simple modules. As described in a previous subsection, this is done with the `@class()` property.

Other useful properties include:

- `@units()` : By appending this property to a parameter, the unit provided (e.g bits,bytes...etc.) is associated with it, and must be

added by the programmer when assigning values (e.g. useful to avoid entering minutes instead of seconds).

- `@statistic()`: Used when applying the *Signals* mechanism to gather statistical information from the simulation (see section 3.2.8 for more information).

- `@display()`: Used to provide information on how NED elements should be displayed during graphical simulation runs (se section 3.2.7).

Module Interfaces

The NED language adopts the concept of interfaces. A module interface can specify certain elements like gates and parameters that must be defined in all modules implementing it. Using module interfaces, submodule types may be specified by string literals assigned to module parameters. This concept is easiest explained by an example:

```
network MyNetwork
{
    parameters:
        string nodeType;
    submodules:
        node[10]: <nodeType> like SensorNode;
    ...
}
```

In the above code, `SensorNode` is a module interface defining all parameters required of a module to be considered a "sensor node". The module type used, depends on which string literal is assigned to the `nodeType` parameter. Valid string literals include all names of modules implementing the `SensorNode` module interface using the `like` keyword.

3.2.4 Programming simple modules

Simple modules are the active components of an OMNeT++ simulation model and are given functionality through programming using the C++ simulation library. To implement a simple module, the user needs to subclass the `cSimpleModule` class, and redefine one or more of the following functions.

- `void initialize()`: Function intended for initialisation code.

- `void handleMessage(cMessage *msg)`: Event routine using messages to represent the events (represented by `cMessage` objects).

- `void finish()`: A function intended for result recording. Called upon successful termination of a simulation.

Another event routine, called `activity()`, is also available in OMNeT++, but is generally not recommended because of scalability issues. All simulation models described in this thesis uses the `handleMessage()` approach. For more information about `activity`, the reader is referred to [19].

The handleMessage() Approach

As mentioned in section 3.2.2, the passing of messages represents the events. More precisely, the events are represented by the reception of an OMNeT++ message, which is an instance of the `cMessage` class (or a subclass of `cMessage`). At reception, the message is handed to the module's `handleMessage()` method. As the method name indicates, `handleMessage()` determines how the message should be treated and thus represents the event routine in OMNeT++. By the principals of discrete event simulation, simulation time only advances in discrete steps between `handleMessage()` calls, and never inside the method itself.

When a message arrives at a module, the idea is to let `handleMessage()` forward it to the appropriate subroutine based on information gathered from that message. The `cMessage` object, representing the message, contains several fields with information that can be used to differentiate between messages. Some examples are:

- *Message kind*: An integer field typically used to identify the type, category or identity of a message (e.g. sequence numbers, data and routing packet differentiation)
- *destination gate*: Gives the ID of the gate that the message arrived on. This is highly useful when implementing a protocol stack, where it can be used to determine which protocol layer the message is coming from.
- Other useful fields: *source module*, *destination module*, *send time*.

The code example below shows a typical implementation of `handleMessage()`

```
Protocol::handleMessage(cMessage *msg)
{
  if (msg->isSelfMessage()){
    handleSelfMessage(msg)
  } else if(msg->getArrivalGateId()== upperLayerIn){
    handleUpperMessage(msg)
  } else if(msg->getArrivalGateId()== lowerLayerIn){
    handleLowerMessage(msg)
  }
}
```

The first `if` statement checks whether the arriving message is a *self-message* (self-messages are used to implement timers and will be further discussed in a later subsection). The second and third statements determines whether the message arrived on an a gate connected to the

upper layer or the lower layer respectively. Based on which `if` condition is met, the message is handed over to the appropriate method.

The handleMessage() approach makes for easy to read code, since there is a single place in the module's code where a newcomer can start reading.

Sending Messages

Messages can be sent directly to other modules or via predefined gates and connections. The following functions are used for this purpose:

- `send()`
- `sendDirect()`

Both are overloaded functions taking two or more arguments.Gates are specified using *gate names* as defined in the NED files.

Implementing Timers

As with most things in OMNeT++, timers are implemented using messages. Since events are only represented by the reception of a message, a timer is implemented by letting the module send a message to itself. The `handleMessage()` method then needs to forward this message to an appropriate subroutine that takes the right action and possibly reschedules the timer. Self-messages are scheduled using the `scheduleAt()` method. In fact, the use of this method is the only thing separating them from other messages. Upon reception, a module can identify a self-message by applying the `isScheduled()` method.The following code initialises and schedules a "hello-timer" of a routing protocol;

```
double interval = par("helloInterval");
helloTimer = new cMessage("hello-timer");
scheduleAt(simTime()+ interval,helloTimer)
```

In the first line, the `par()` function returns the value of a parameter named "helloInterval", previously given a value in a NED or INI file, which is then assigned to the `interval` variable. The second line then creates the timer, represented by a `cMessage` object, and names it "hello-timer". Finally, the timer is scheduled at `simTime()+interval`, where `simTime()` is a function returning the current simulation time.

Representing network packets

Network packets in OMNeT++ are represented by instances of the `cPacket` class. In addition to the fields inherited by its super class (`cMessage`), `cPacket` provides *bit-lenght* and *bit-error-flag* fields. An important concept from the world of communication systems is the encapsulation of packets. In `cPacket`, this is implemented by the following function:

- `void encapsulate(cPacket *packet)`

- `cPacket *decapsulate()`

The encapsulation of one packet into another using `encapsulate()` increases the bit-length field accordingly, and only one packet can be encapsulated at a time. When retrieving the encapsulated packet, `decapsulate()` changes the bit-length back to its original value. The `cPacket` class does not support encapsulating more than one packet. Adding this and other functionality can by done by sub-classing it [19].

User defined network packets are usually sub-classed from `cPacket`. Several different packet types are often required to model the full protocol stack of a communication system, and writing them all in C++ with the necessary setter and getter function can be a tedious task. OMNeT++ provides a simple and compact syntax with *message definitions* to lighten the work load. As an example, the message definition of a simple network layer packet, `NetwPkt` can be written as:

```
packet NetwPkt
{
int destinationAddress;
int sourceAddress;
int ttl = 1;
unsigned sequenceNum = 0;
}
```

The content above is stored in a .msg file. When executing a simulation run, OMNeT++ will generate full fledge C++ classes on the fly using its own *message compiler*. The resulting classes will be `NetwPkt_m.h` and `NetwPkt_m.cc` and will contain `set` and `get` methods for all the fields specified. By default the generated classes will inherit `cPacket`, but it is also possible to extend other packet classes using the `extends` keyword:

```
packet MyRoutingPkt extends NetwPkt
{
int hopCount;
int finalDestination;
int initialSource;
}
```

Initialisation And Finalisation

As described in section 3.1, all discrete event simulators need a way of providing the FEL with the initial events. In OMNeT++, this is done by calling the `initialize()` method for all simple modules. All initialisation code should be placed in this method because when constructors are called, the simulation model is still being set up, and required objects may not be available. Some uses for the `initialize()` method are:

- Parsing parameters from the NED and INI files
- Initialise timers

- Initialise statistical counters
- Register simulation signals (see section 3.2.5)

Upon successful termination of a simulation, the `finish()` function is called. It is intended for statistics collection only, and the programmer must be careful not to insert any cleanup code here. This task should be done in the destructor.

3.2.5 Simulation Signals

Simulation signals, or signals for short, is a built in notification mechanism that first appeared in version 4.1 of OMNeT++. Some uses are [19]:

- expose statistical information from the simulation model without specifying whether or how it should be recorded
- receive notifications when certain changes occur in the simulation model at runtime, and act accordingly
- implement "publish-subscribe" style communication among modules

Signals are emitted from simple modules and can carry arbitrary data (even object pointers). *Listeners* located anywhere in the module hierarchy can subscribe to these signals by name, and receive notifications whenever a signal is emitted. The subscribing module can then access the content of the variable emitted by the signal.

Registering Signals

The signals emitted from a simple module needs to be registered with OMNeT++ before use. This is done in the `initialize()` function using the following statement:

```
simsignal_t signalId = registerSignal("signalName");
```

Though identified by names given by the programmer, internally, signals are represented by numerical identifiers type defined as `simsignal_t`. The `registerSignal()` function returns an integer to be used whenever referencing this signal. The mapping of signal names to identifiers is global.

Emitting Signals

Signals are emitted using the `emit()` function which takes two input arguments; the signal identifier, and the variable to be exposed.

```
emit(signalId, variable)
```

Actually, `emit()` is a family of overloaded functions so `variable` in the statement above can be of any type that can be cast into one of the folowing types:`long`, `double`, `simtime_t`, `const char*` or `cObject *`, where `simtime_t` is a high resolution integer representing the simulation time.

Subscribing to Signals

Listeners are instances of classes that extend the `cIListener` class [19]. Subscribing to signals is done by calling the `subscribe()` function that takes two arguments; the signal ID and a pointer to a listener object:

```
subscribe(signalID, pointerToListener);
```

The `signalID` is obtained by calling `registerSignal("signalName")` as described earlier. For convenience, an overloaded version of subscribe takes the signal name directly:

```
subscribe("signalName", pointerToListener);
```

How listener classes are implemented is out of the scope of this thesis. However, when using the signals mechanism to expose statistical information from the model, OMNeT++ will automatically add listeners based on `@statistic` properties in the NED declarations. This concept is further discussed in the result recording section (3.2.8).

3.2.6 Configuring Simulations

An important design principle in OMNeT++ is the separation of experiments from models, where a model is defined by its NED definitions together with the corresponding C++ code. Experiments are defined in the INI files. It is assumed that a large number of different experiments needs to be conducted on a single model before a conclusion can be drawn. For this reason, although parameters can be hard-coded in the NED files, it is recommended that most parameters are set using INI files.

INI file structure

An INI file consist of different sections, each representing an individual experiment:

```
[General]
...
[Config Scenario1]
...
[Config Scenario2]
...
```

Only the `General` section, is mandatory and other sections are referred to as *configuration sections*. All parameters set in `General` are inherited by the configuration sections, and should contain parameter settings common to all experiments. In addition, configuration sections may inherit parameters from each other using the `extends` keyword as follows:

```
...
[Config Scenario1]
...
```

```
[Config Scenario2]
extends = Scenario2
...
```

File inclusion

Some simulation models may require that a large number of parameters must be set in the INI file. To avoid large configuration files, parameter settings can be partitioned into logical units and included in the main INI file using the `include` keyword. As an example;

```
[General]
...
include basicParameters.ini
...
```

includes all parameters assigned in `basicParameters.ini`. The only exception is that parameters already set in a section cannot by changed by an inclusion.

Assigning parameters

Parameters are assigned values in INI files by specifying the full hierarchy path from the NED definitions. As an example, the following statement sets the `allowAck` parameter of a module called `MyMAC`;

```
MyNetwork.MyNode1.MyMAC.allowAck = true
```

where `MyNetwork` is the top module, and `MyNode1` is a compound module containing `MyMac`. In a full scale network simulation consisting of a large number of nodes, setting parameters on a per node basis would be a tedious task. For this reason, statements can be generalized using wildcard patterns. To set the same parameter for all nodes in the simulation the statement above can be replaced by:

```
**.MyMAC.allowAck = true
```

It is important to note, when using wildcards that, the order of entries matter. If a parameter name matches several wildcard patterns, the first pattern will be used.

3.2.7 Running Simulations

Both individual and batch runs can be executed directly from the *Integrated Development Environment* (IDE) or from the command-line. OMNeT++ includes two user interfaces; the `Tkenv` graphical user interface, and the `Cmdenv` command-line interface used for batch runs. This section briefly introduces these user interfaces and shows how simulations can be executed from the command-line.

Tkenv

The `Tkenv` graphical user interface is highly useful for demonstration and debugging purposes. It allows interactive runs where the user can inspect the occurrence of each event in a step by step fashion, or watch an animated representation of the entire simulation. At any instance of simulation time the contents of individual objects can be explored.

Cmdenv

Batch runs are performed using the `Cmdev` command-line user interface.

Executing Simulations from the Command-Line

When scripting is involved, executing simulations from the command-line is essential. Under linux, the simulation executable can be executed using the standard "./" prefix followed by the executable's name:

```
$./MySimulation
```

To specify a user interface, which configuration files to use etc., command line options are used. Table 3.1 shows the different command line options used to run simulations in this project. As an example;

```
$./MySimulation -u Cmdenv -f myConfig.ini -c experiment1
```

executes all runs specified in the `experiment1` configuration section in the `myConfig.ini` file, using the `Cmdenv` user interface.

Option	Description	Default value
-u	selects a user interface	Tkenv
-f	specifies the INI file	omnetpp.ini
-c	specifies the configuration to be run	`General`
-r	specifies the simulation runs to be executed	0 with Tkenv `all` with Cmdenv

Table 3.1: OMNeT++ command line options

3.2.8 Result Recording

When a simulation is terminated successfully, OMNeT++ outputs statistical information gathered throughout the simulation run into *vector* and *scalar* files. For time series data like round trip and queuing times, output vectors are used, while output scalars produce summary results (e.g. sum, mean, standard deviation, minimum and maximum values etc.). Generally, there are two ways of collecting and recording statistics:

1. Using the simulation signal mechanism
2. Directly from the C++ code, using only the simulation library.

With the second approach, sometimes referred to as *direct result recording*, results are collected as member variables in simple modules. Special functions are then used to record these variables in the `finish()` function. A major drawback of hard-coding result recording in this way, is that the simulation model has to be reprogrammed and recompiled whenever a different set of statistics is required. While designing the simulation model, it is often not possible to foresee all experiments needed to be performed, and accordingly, which statistics will be required. For this reason, only result recording using the simulation signals mechanism was used while designing models in this thesis. Result recording using signals is described next.

Result recording with simulation signals

The simulation signals mechanism described in section 3.2.5 provides a useful way of collecting and recording statistics from the simulation model. Signals can be emitted to expose statistical information from the model without specifying how or whether it should be recorded. Listeners may be added in custom made modules dedicated to statistics collection, or generated automatically by OMNeT++ based on configuration. The latter approach was heavily made use of while working on this project and will be described in the following.

Emitted signals propagate on the module hierarchy up to the root (`network` module). As a result of this, listeners registered at a compound module can receive signals from all modules in its sub-module tree. To record simulation results based on the signals mechanism, the programmer ads one or more `@statistic` properties in a module's NED difinition. As an example, consider the following declaration of a statistic, recording the average RSSI value measured by nodes in a wireless network:

```
@statistic[statRSSI](source="rssiSignal";record=mean );
```

The `statRSSI` entry within the square brackets gives the name of the statistic. If not specified otherwise, this name will also be the title of the statistic written in the result file. The `source` property value specifies the signal by name, and `record` determines how the signal shall be recorded. If the above statement is placed at node level, the result file will include entries for each node. Placing the statement on network level however, would result in a single RSSI value averaged over all RSSI measurements made by the nodes in the network.

3.3 MiXiM

MiXiM is a model framework that extends OMNeT++ with support for mobile and wireless network simulation. It provides several different ways of modelling the wireless channel and offers a rich protocol library primarily focused on the MAC layer. The name MiXiM, underlines that this is a mixed simulator that builds on previously developed

model frameworks. MiXiM is updated regularly with new models and functionality. In this project MiXiM version 2.2.1 (version 3.2 mas released in March 2013).

3.3.1 Modelling Concepts

MiXiM uses the module interface concept extensively, so that most modules mentioned in the following subsections can be of any type implementing the corresponding interface. For example; the `mobility` module can be of any type implementing the `IMobility` module interface and the `netwl` (network layer) module can by of any type implementing the `IBaseNetwLayer` module interface. Designing custom made modules to work with MiXiM is quite straight forward from the NED definition perspective. Because the module interfaces provide clear instructions on what is needed, the programmer only needs to copy the content of the corresponding interfaces and add the desired parameters. Implementing simple modules is mainly done by extending different base classes (e.g. BaseNetwLayer, BaseMacLayer etc.). However, when designing PHY layer modules, it is often sensible to subclass more specialised implementations to achieve compatibility with other MiXiM models. As will be shown in section 3.3.3, not all MiXiM models have a corresponding NED definition.

In the following; the most important modules used when building a MiXiM network is introduced.

Network Level

In addition to node modules, all MiXiM networks must include the following two submodules:

- `World`
- `ConnectionManager`

The `World` module sets the simulation *playground* through gathering parameters that define the dimensions of the network, whether to use 2D or 3D etc [17]. The `ConnectionManager` module sets up, and manages connections between wireless nodes that are within a certain distance, referred to as the *interference distance*, of each other. A MiXiM connection does not mean that the nodes can communicate, but rather that connected nodes are influenced by each others signals represented by OMNeT++ messages. This concept is essential to understand and will be further discussed in section 3.3.2.

MiXiM nodes

Most MiXiM nodes include the following submodules:

- `mobility`: Mandatory module that define the current position and movement pattern of nodes.

- `arp`: Utility module used by modules representing the network and mac layers for address resolution.
- `appl`: Module representing the application layer:
- `netwl`: Module representing the network layer.
- `Nic`: Compound module containing submodules representing the MAC and physical layer.

Through the mobility module, the network topology is defined. There are a range of different mobility modules available, all of which has to specify the initial position of nodes. For a fixed topology the `StationaryMobility` module type can be used. The `arp` module translates network layer addresses into mac layer addresses and vice versa. Currently, the only available `arp` module type, called `ArpHost` uses node indexing to represent both address types. This means that the `arp` module assumes that the nodes are declared as a vector of NED modules. MiXiM mostly focuses on the MAC and PHY layers of wireless networks, while the higher layers are better represented in other model frameworks. For traffic generation, MiXiM provides the `BurstApplLayer` and `TrafficGen` module types for broadcast traffic, while unicast traffic is only supported by the `SensorApplLayer` module. Support for routing is mainly given by the `Flood`, `ProbabilisticBroadcast` and `WiseRoute` module types. The latter was used in the simulation models for this project and will be further described in chapter 4. The modules representing the physical and MAC layers of the protocol stack is usually grouped into a compound Network Interface Card (NIC) module. The reason for this, given in [17], is that the design of these layers is usually tightly coupled and very specific for different communication techniques.

3.3.2 Connection Modelling

Modelling connections between nodes is much more challenging in a wireless environment than with nodes that communicate through cables and wires. In wired simulations, the cable between two nodes is easily modelled by an OMNeT++ connection together with an appropriate channel module. In wireless simulations, however, the transmission medium to be modelled is the air, and it is not as easily modelled by a single OMNeT++ connection.

Theoretically, all nodes in a wireless simulation, assuming that they are transmitting on the same frequency, interfere with each other. However, if nodes are sufficiently far apart, the interference imposed by one node on the other becomes negligible. To avoid computational complexity, MiXiM lets the user define a *maximum interference distance* through configuring the ConnectionManager module. Connections are then dynamically established between nodes that are within this distance of each other. It is important to note that MiXiM connections does not necessarily mean that nodes can communicate, but rather that connected nodes have to

consider each others messages. The task of deciding if a message should be considered as interference or a receivable signal is given to the PHY layer module in the receiver node.

Configuring the ConnectionManager

The following four parameters are used to calculate the interference distance between nodes:

- `pMax`: maximum sending power used in the network [mW]
- `sat`: minimum signal attenuation threshold [dBm]
- `alpha`: minimum path loss exponent
- `carrierFrequency`: minimum frequency of the signals used [Hz]

From reading the C++ implementation of MiXiMs `ConnectionManager` the interference distance is calculated by:

$$\text{interferenceDistance} = \left(\frac{\text{pMax} \cdot \lambda}{16\pi^2 \cdot \text{sat}} \right)^{1/\text{alpha}} \tag{3.1}$$

where λ is the wavelength derived from the frequency parameter.

3.3.3 The PHY Layer and Channel Modelling

The PHY layer is a core module in MiXiM. In addition to responsibilities like the modelling of transmission and reception of physical signals, and providing services to the MAC layer, it is also responsible for modelling the channel effects. The class graph in figure 3.3 shows its most important components. The `BasePhyLayer` provides the basic functionality of a physical layer and is directly connected to the MAC layer module through OMNeT++ connections. PHY layer modules in different wireless nodes communicate by passing `AirFrame` messages to each other, each containing a `Signal` object that provides a physical representation of the wireless signal. Upon the reception of an `AirFrame`, the receiving module first ads attenuation effects to the signal by using filters known as `Analogue Models`. The task of determining whether or not the attenuated signal can be received, or if it should be considered as noise/interference, is then given to an entity known as the `Decider`. `Analogue Models` are responsible for simulating the effects of the channel (e.i. path loss, shadowing and fading), while the responsibilities of the `Decider` include the modelling of demodulation (bit error calculation) and the implementation of channel sensing. The programmer can choose between different `Analogue Models` and `Deciders` available and apply them as xml parameters to the appropriate PHY layer module. Both `Analogue Models` and `Deciders` are pure C++ classes and will be further described in sections 3.3.4 and 3.3.5 respectively.

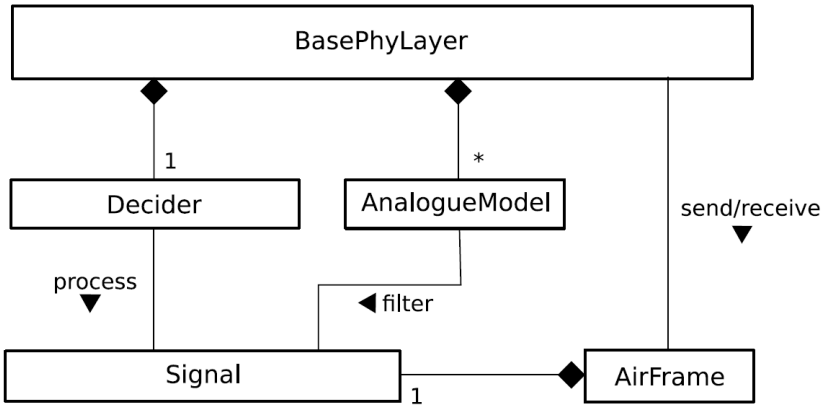

Figure 3.3: MiXiM physical Layer class-graph [17]

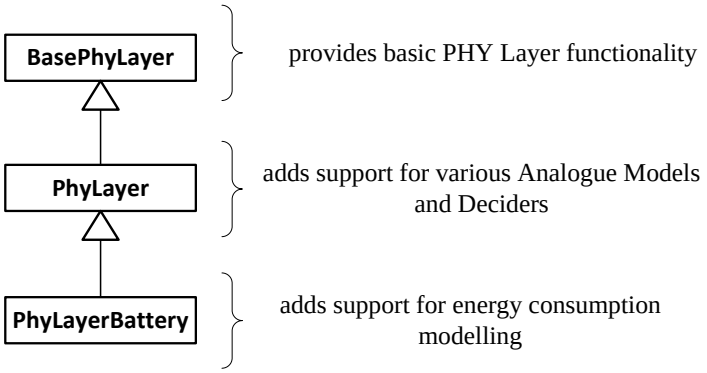

Figure 3.4: MiXiM PHY layer inheritance diagram[17]

As mentioned, the `BasePhyLayer` module provides basic PHY layer functionality. Interfaces are implemented to ensure proper communication with the MAC layer and to allow the use of `AnalogueModels` and `Deciders`. Different transceiver states (e.i. TX, RX and SLEEP) are kept track of and represented as a state machine. The `BasePhyLayer` also keeps track of all active `AirFrames` to allow interference calculations and channel sensing. Each `Analogue Model` or `Decider` requires its own initialisation procedure. Because the `BasePhyLayer` is designed to be a general abstraction of a PHY layer, the implementation of such procedures is given by its subclasses. An inheritance diagram showing the most important PHY layer modules are given in figure 3.4. The `PhyLayer` module extends `BasePhyLayer` with support for most `AnalogueModels` and `Deciders` available in the MiXiM, and it is further extended by `PhyLayerBattery` to support energy consumption modelling.

3.3.4 Analogue Models

As mentioned, `Analogue Models` are filters that add the effects of the channel to the transmitted signal. The signal object contained in the received `AirFrame` provides a mapping of the transmitted signal power (usually a rectangular function of time). To apply the channel effects, attenuation mappings are added by the `Analogue Models` that define the attenuation factors for the signal. By multiplying the signal and attenuation mappings a mapping of the received signal is gained.

Mappings can have several dimension (e.g. time, frequency or space). An attenuation mapping for the time dimension contains "key entries" for the discrete points in time where the attenuation factors are defined (depending on the Analogue Model implementation). When retrieving the received signal level for a specific time interval the resulting value is interpolated.

The following available `Analogue Models` can be used with the `PhyLayer` module or any of its submodules:

- `SimplePathLossModel`: calculates mean path loss based on distance
- `LogNormalShadowing`: ads log-normal 0 mean attenuation factors based on an interval
- `JakesFading`: implements Rayleigh fading according Jakes model [ref]
- `PERModel`: based on a PER parameter (PER $\in [0,1]$) the attenuation factor for the whole signal is randomly set to either 0 (packet will be lost) or 1 (packet will be received)

Several `Analogue Models` can be used simultaneously. In fact, both `LogNormalShadowing` and `JakesFading` are meant to be used together with `SimplePathLossModel` to give the appropriate effect.

3.3.5 Deciders

Unlike `Analogue Models`, only one `Decider` can be used by the PHY layer module. The following tasks are given to the `Decider`

- Decide whether the incoming `AirFrame` should be considered as a receivable signal or noise/interference
- Decide whether the signal can be correctly received
- Implement channel sensing

How these tasks are performed depends on the implementation of `Decider` used.

Error Calaulations Based On SNIR

As mentioned in 3.3.4 the analogue models define attenuation factors at discrete points in time (or other dimensions) for incoming signals. Error calculations are performed for each interval between two such discrete points. Similarly more discrete points are added by the decider to represent interference. Though the mappings from `AnalogueModels` and interference calculations are time varying functions, the thermal noise is represented by a constant.

3.3.6 Modelling Energy Consumption

MiXiM allows the modelling of energy consumption through `battery` modules. These modules require a special implementation of the PHY layer called `PhyLayerBattery`, and uses a separate module dedicated to the gathering of energy consumption statistics referred to as `BatteryStats`. Because of issues concerning scalability and implementation, further discussed in section 3.4, `battery` modules were not used in any models developed in this project, and is only mentioned here for completeness.

3.3.7 Result Recording

Statistics collection in MiXiM follows the direct result recording approach as described in section 3.2.8. This means that results are collected as member variables of simple modules and recorded in the corresponding `finish()` functions using the simulation library. Collecting results from deciders is done in the `finish()` function of PHY layer modules.

Part of the reason why MiXiM does not apply the simulation signals approach to result recording might be because many of the available modules was implemented before it was made available.

3.4 The Simulation Models Used

This section describes how different modules, both self made and taken from the MiXiM protocol library directly, were implemented and put together to form the simulation model used in this project. The last subsection (3.4.5) explains how the modules were configured for result recording.

3.4.1 The Network Module

The implemented network module, called `MittNettverk`, extends the `BaseNetwork` MiXiM module that provides the basic components needed for a MiXiM network (e.i. world and ConnectionManager). `MittNettverk` simply adds a vector of nodes, type defined as `SensorNode`, and a parameter to specify the number of nodes in a given simulation. It also adds several `@Statistic` properties for result recording further discussed in section 3.4.5. For the full implementation of `MittNettverk` see appendix D.1.

3.4.2 The Sensor Node Module

This subsection describes the basic node model, named `SensorNode`, developed and used in the different simulation scenarios in this project. The model is based on a MiXiM module called `Host80215_2400MHz`, that defines a node using an IEEE 802.15.4 transceiver for wireless communication at 2.4GHz. The MAC layer uses a non beacon enabled CSMA protocol as specified by the standard and the transceiver is modelled according to the CC2420 Texas Instruments chip. This particular model has been independently evaluated using a wireless sensor network testbed in [21]. However, the model also includes a transport layer module and modules for battery consumption modelling not needed for the purposes of this project.

To enhance the performance of the model, these modules were consequently removed. Simulation test runs performed with the original MiXiM model proved to be very time-consuming when the number of nodes were high. In [8], experiments with large networks (from 125 to 10,000 nodes) using OMNeT++ were performed to assess the influence of various simulation components on the performance of the model. A battery model, similar to the one used in MiXiM, was shown to degrade the performance by two to three orders of magnitude. Another reason why the battery module was removed is because of the way the results are gathered. In this project, only energy consumption due to data traffic is investigated. When routing is involved, a way of filtering out the energy consumed by routing is needed. The battery model in MiXiM does not provide any simple way of doing this, however, as will be discussed in section 3.4.5 , the signals mechanism provides an elegant solution to this problem. As transport layer functionality is not investigated in this project the transport layer module is not needed.

Figure 3.5 shows the composition of the modified node named `SensorNode`. All the modules depicted are place holders for actual module types that are assigned via string parameters. The different simulation scenarios in this project require various implementations of the network layer module, but other modules uses the same type definitions throughout. The following submodule types are common to all versions of `SensorNode` used:

- `StationaryMobility`: `mobility` module for a fixed network
- `ArpHost`: simple arp module implementation for address resolution
- `SensorApplLayer`: application layer module for unicast traffic
- `modCC2420Nic`: self modified version of MiXiM's `Nic802154_TI_CC2420` module

These module types are basically the same as the ones implemented in the `Host802154_2400MHz` (excluding the transport layer and battery modules). The motivation for altering `Nic802154_TI_CC2420` was that the original module includes parameters to work with the battery module. For more information on how the module definition of `SensorNode` is designed see appendix D.2.

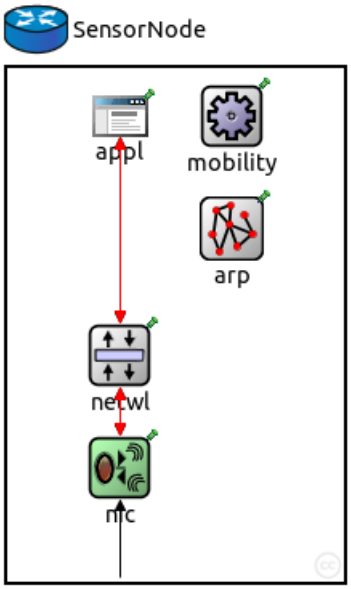

Figure 3.5: Module composition of SensorNode

3.4.3 Physical and MAC Layer Modules

The PHY and MAC layers of `SensorNode` are grouped in the compound `modCC2420Nic` module as shown in figure 3.6. As mentioned earlier the MAC module models a non beacon enabled CSMA protocol as specified by

the standard and is connected to the PHY layer via OMNeT++ connections. The PHY layer implementation is described in more detail in the following.

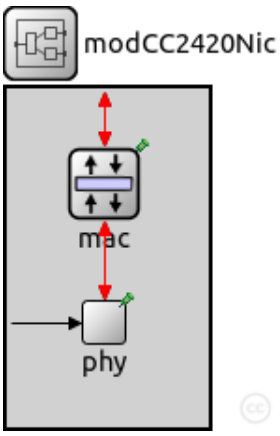

Figure 3.6: Module composition of modCC2420Nic

Physical Layer

The PHY layer module is basically the same for all physical layer implementations in MiXiM. The defining functionality of individual PHY layers is mostly implemented by the `Decider`. The `Decider802154Narrow` class is designed to model several different PHY layers as defined in the IEEE 802.15.4 standard by providing models for individual modulation techniques. In this project O-QPSK is used for all simulation scenarios, except for the Connectivity scenario in section 4.1.3 which uses a different decider altogether.

The decision of whether a packet can be received or not is made in two parts; first, error calculations are carried out on the bits in the start frame delimiter (SFD) to see if the frame can be synchronized, and then, the rest of the frame is checked for bit errors. The synchronization test provides a way of identifying why a packet is dropped, and it also plays an important part in the modelling of channel sensing. The decider only reports the medium as busy if an `AirFrame` that passed the synchronization test is currently on the channel. Channel sensing is performed on the behalf of the MAC layer by measuring the RSSI level in a given interval. The RSSI level is given as the sum of signal, thermal noise and interference levels, where the thermal noise is a constant.

All error calculations are based on the signal to noise plus interference (SNIR) level of an incoming packet. When several packets overlap in time, the first packet to arrive is considered as the signal, while the others are defined as interference. The `Decider802154Narrow` uses following formula for bit error calculations when O-QPSK is enabled (as read from the API):

$$P_b = \frac{8}{15} \cdot \frac{1}{16} \sum_{k=2}^{16} e^{20 \cdot SNIR \cdot (\frac{1}{k} - 1)} \tag{3.2}$$

which is as specified by the IEEE 802.15.4 standard 2.1.

3.4.4 Network Layer Modules

In this section the various network layer modules used in the different scenarios in this project are described. The `DummyRoute` module was taken directly from the MiXiM protocol library while the `RSSIRouting` module is a self made module based on a pre-existing MiXiM module called `WiseRoute`. `ClosetRouting` is an entirely self made module.

Dummy Route

For simulation scenarios that does not involve routing the `Dummy Route` module is used for efficiency. It makes sure that network addresses are translated into MAC address, but provides no routing.

RSSI Routing

`RSSIRouting` is a self made routing protocol based on the `WiseRoute` network layer module available in the MiXiM protocol library. It adopts most of its code directly from `WiseRoute`, but functionality was added to meet the following requirements needed for the scenarios in this project:

1. Best path is found using hop count as metric
2. Only links with quality higher than a certain controllable threshold is considered
3. Protocol messaged shall not interfere with data traffic

WiseRoute is a simple protocol that builds a routing tree from a central point (the sink). The sink initialises the tree building process by broadcasting a *route flood* message. All nodes that receive the route flood register the sink as parent in the routing tree and rebroadcast the message. The sink continues to broadcast route floods at specified intervals throughout the simulation. Sequence numbers are used to avoid routing loops, so that outdated flood messages can be identified and discarded immediately. To avoid routes being installed with poor quality links an RSSI threshold can be set. Route Flood messages received on a link with an RSSI level below this threshold is also discarded.

A general problem with `WiseRoute` is that routes are only installed once, and never updated. Collisions occurring during the route flooding process may lead to situations where routes are installed wrongly or not at all. In dense parts of a network collisions happen more frequently and slows down the propagation of route floods. By propagating through less dense parts a network, a route flood may arrive earlier at a node having traversed a higher number of hops than necessary. In RSSIRoute this problem was combated by expanding the `WiseRoute` routing table to include hop count as metric and by introducing code for updating routes whenever a better

one is available. A flow chart illustrating how nodes process incoming route floods in RSSIRoute is given in figure 3.7.

A simple way of making sure that protocol messages does not interfere with the data traffic, is to stop the dissemination of floods before any data packets are transmitted. This is sufficient for the scenarios in this project. Only fixed networks are considered and the quality of links does not change in the course of the simulations. A parameter, `maxFloods`, was introduced to limit the number of route floods broadcasted by the sink.

For the full implementation of RSSIRouting see appendix D.3.2

Closest Routing

Closest Routing is a simple self made routing protocol designed be used in the Line Topology Scenario (see section 4.1.4) only. In a line topology where nodes are addressed incrementally from left to right this module makes sure that packets are forwarded through all nodes between the sender and the sink. For the full implementation of `ClosestRouting` see appendix D.3.1.

3.4.5 Result Recording Configurations

As mentioned in xxx MiXiM uses the direct result recording approach for statistics collection. Boolean parameters are used to switch result recording from individual modules on or off. A downside to this is that some modules create large amounts of statistics to be able to fit the needs for most simulations. When simulating large networks with high numbers of nodes the corresponding result files may become very large even though the user only needs a few variables per node. For this reason the signal mechanism was used to record most of the results.

In the simple modules, signals were registered to emit the values of different variable of interest. `@Statistics` properties were then added to the NED definition at both node and network level, and configured to set in the INI file. This made it possible to collect results on a per node and parameter basis. Adding `@Statistic` properties at the network level makes it possible to gather statistics as a sum of all nodes in the network (e.g. total number of transmissions, retransmission, acknowledgements etc.). For more information on how the `@Statistics` properties were added and configured see the individual NED definitions of `SensorNode` and `MittNettverk` in appenix D.2.1 and D.1 respectively.

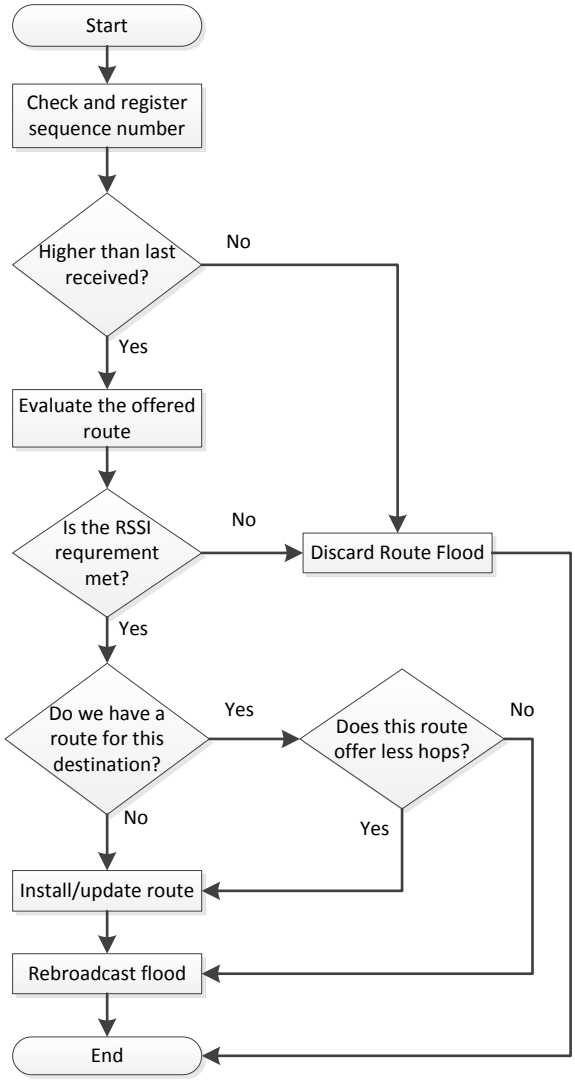

Figure 3.7: Flow Chart illustrating the route flood handling procedure in RSSIrouting

3.5 Scripting and Other Tools

This section briefly presents how scripting and other tools were used while working with this project. In addition to Matlab and Python scripts, Microsoft Visio 2010 was used to produce flow charts and diagrams presented in this document.

3.5.1 Python Scripting

Scripts written in the python programming language was developed for both running the simulations and processing the results presented in this document. Regular expression where used to retrieve information from the OMNeT++ generated output vector and scalar files, and the processed results were formatted as MATLAB vectors. Processing included energy calculation based on current consumption characteristics given in the CC2420 radio chip, and averaging results over a number of repeat runs. Scripting was also used to change the configuration files between simulation runs to avoid tedious manual editing. Various Python scripts used can be found in appendix B.

3.5.2 Matlab

All result graphs presented in this document was produced using Matlab. Additionally, analytical results where calculated using self made Matlab scripts and functions. Matlab code can be found in appendix C.

Chapter 4

Simulations and Analysis

In this chapter, the different simulation scenarios are described and results from simulations are presented and analysed. Section 4.1 introduces the scenarios and discusses the most important parameters used in the simulations. Section 4.2 present the results from the simulations and also includes analytical results from applying the equations presented an developed in chapter 2. Results are individually analysed when presented, but section 4.3 aims to discuss them in a broader context.

4.1 Scenarios

This section describes the various investigated simulation scenarios. First, the most important parameters are discussed.

4.1.1 Parameters and Settings

In this subsection the most important parameters used in the different simulation scenarios are presented. The same parameters apply for the analytical part as well. If not stated otherwise the parameters listed in table 4.1 are used. The connectivity scenario presented in section 4.1.3 used the simplified *disk model* approach that will be described at the end of this subsection.

The Path Loss Exponent

As discussed in section 2.3.2 values for the path loss exponent usually varies from 2-6 depending on the environment. In [11] the authors refers to measurement carried out in a factory environment where α was found to be in the range of 2-3, and a typical value for α in both indoor and outdoor environments is referred to in [28] as 3. The path loss exponent was accordingly chosen to be 3 for the various simulation scenarios.

Packet Sizes

The packet size parameters presented in table 4.1 are given as the total frame length as seen at the PHY layer including headers. A PSDU of 20

bytes was used for data packets and the ACK packet PSDU was set to 5 bytes as described in the standard [20]. PHY header length was set to 6 bytes.

Parameter	Value	Unit
Path Loss Exponent (α)	3.0	-
Thermal Noise	-95.53	dBm
Data Packet Size (L_{DATA})	26	bytes
Ack Packet Size (L_{ACK})	11	bytes
Transmission Output Power (P_{tx})	0	dBm

Table 4.1: Simulation Parameters

Determining the thermal noise parameter

As described in section 2.1.2 the receiver sensitivity is defined as the lowest input power for which the PER is no higher than 1%, and where only thermal noise is considered. With the `SimplePathLossModel` as the only Analogue Model, and a path-loss exponent of 3, the CC2420 receiver sensitivity of -95dBm should result in a range of $d = 67.63$m. The MiXiM model of the CC2420 radio chip has the thermal noise parameter set to -110 dBm by default. A simulation with this model reveals the received power can drop to about -110 dBm before the PER reaches 1%. To be compliant with the sensitivity, as specified by the data sheet, the choice was made to adjust the thermal noise so that the model yields a PER of 1% at an input power level of -95 dBm.

Using the node model as described in section 3.4.2, a two node scenario was set up. Nodes where placed 67.63 meters apart, giving a received input power of -95 dBm, and 10000 packets where transmitted to the receiving node. In each consecutive simulation run the thermal noise parameter was increased in steps of 0.01 dBm until the PER reached 1%. Averaged over 20 repeat runs, a PER of 1.01% was achieved with the thermal noise parameter set to -95.53 dBm

- If not stated otherwise, the thermal noise parameter is set to: `**.node[*].nic.phy.thermalNoise = -95.53 dBm`

The Disk Model Approach

The Connectivity Simulations Scenario in section 4.1.3 is configured to follow the Disk Model approach. This is a simplification that imply that a transmission is either 100% successful or fails completely [13], depending on the transmission radius defined by the receiver sensitivity. All packets with a corresponding signal power level above the receiver's sensitivity threshold is successfully received while signal levels below this threshold imply packet loss. To be consistent with the CC2420 transceiver the sensitivity was set to -95dBm. When this model is used retransmissions are unnecessary and is therefore not used. The Disk Model is achieved by using the `SNRThresholdDecider` and by configuring the PHY layer

module to not simulate thermal noise (`useThermalNoise = false`). For more information about the `SNRThresholdDecider` the reader is referred to [19].

4.1.2 Two Node Scenario

This simple two node scenario investigates how the number of transmissions needed to transmit a packet successfully relates to node range. This is done by simulations with increasing node separations and analytically using equations developed in section 2.5.

Simulation Scenario Description

In the first simulation run, the two nodes were placed in the topology area separated by an initial distance d_0. The sender then transmitted 10000 to the receiver. In each consecutive run, d_0 was increased by a smaller distance Δ until the separation of the nodes reached a point where all packets are lost. Figure 4.1 shows the progress of the simulation.

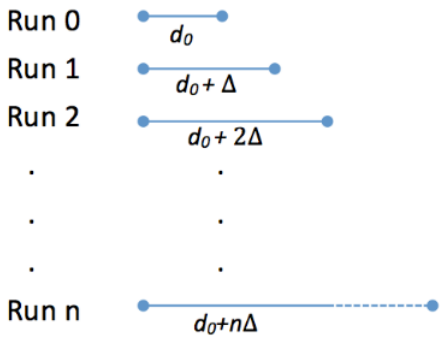

Figure 4.1: TwoNode simulation run progress

The simulation scenario was run twice with the parameter for the maximum number of retransmissions allowed set to 3 and 7 respectively. The standard allows values in the range of 0-7, where 3 is the default value [20].

4.1.3 Connectivity in Random Networks

This simulation scenario was used to investigate connectivity in a fixed random topology network. The goal was to find out which node densities are needed to gain connectivity with different output power configurations. The nodes in this scenario uses the `SNRThresholdDecider` and were configured to follow the Disc Model approach as described in section 4.1.1.

The sender node and sink was placed at fixed positions separated by 100 meters in a 300x300 topology area as shown in figure 4.2. A number of nodes according to a given node density, λ, were generated and randomly

distributed across the topology area following a uniform distribution. Depending on the node density, the number of nodes in a simulation run is given by:

$$\text{numNodes} = \lfloor A \cdot \lambda \rfloor, \tag{4.1}$$

where

- $\lfloor x \rfloor$ denotes the floor function of x
- λ is the node density $[\text{nodes}/\text{m}^2]$
- A is the topology are $[\text{m}^2]$

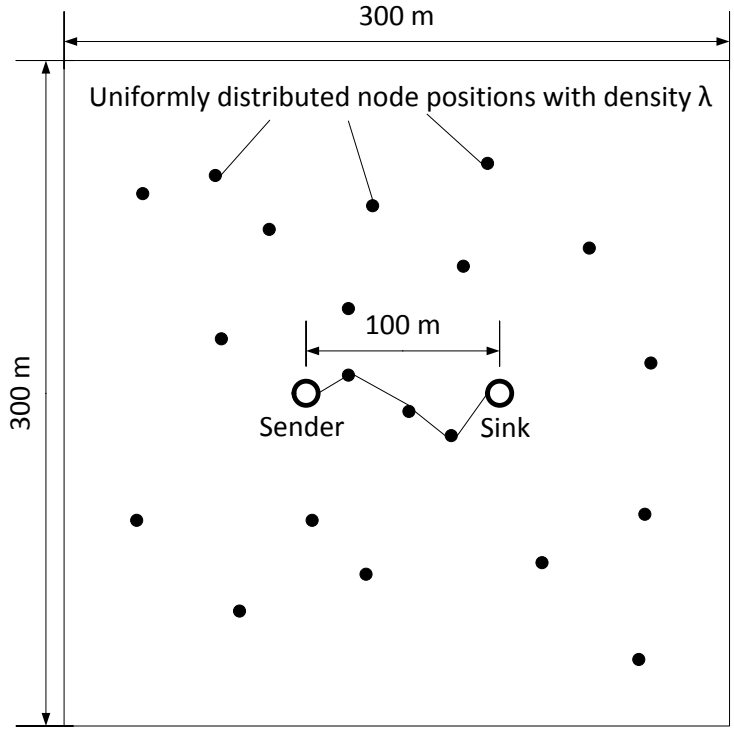

Figure 4.2: Random topology scenario with a sender that transmits packets to a sink node 100 meters away.

To prevent protocol messages from interfering with data traffic the sender starts transmitting data packets after all routes have been set up. Simulation test runs, described later in this section, show that 40 route flood messages are enough to make sure that the routes are set up properly. Because this scenario uses the disk model; all transmitted data packets will arrive at the sink if a route exists. Basically, connectivity and the number of hops needed to reach the sink can be assessed by investigating the routing table of the sender. To test that routes were set up properly 10 data packets were transmitted to the sink. The simulation progresses as follows:

$t = 0s$: Dissemination of route floods starts

$t = 40s$: Last route flood is broadcasted

$t = 120s$: Transmission of data packets start

The simulation is ended when the sender has no more packets to send.

Finding the right number of floods

As described in section 3.4.4, collisions may prevent that the best routes are found if only one route flood is broadcasted. How many floods are needed depends on the network topology. To find a "safe" number of route floods that is sufficient for most networks, simulation tests were performed. Random networks with different node densities were set up as described in this section and simulated using different numbers of route floods. Each random network was generated using the same *seed* as input for the random generator in order to produce the same topologies while varying the number of floods. For both high and low density networks ($\lambda = 0.001$ and $\lambda = 0.005$) 40 route floods was enough to produce the same routes in all simulation runs.

Random Topology Generation

The random topology as shown in figure 4.2 is created by the following INI file configuration.

```
#Manualy placing sender and sink
**.node[0].mobility.initialX = 100m
**.node[0].mobility.initialY = 200m
**.node[1].mobility.initialX = 200m
**.node[1].mobility.initialY = 200m

#Randomly drawing node positions for other nodes
**.node[*].mobility.initialX = intuniform(0, 300)*1m
**.node[*].mobility.initialY = intuniform(0, 300)*1m
```

Parameter Studies

Eight simulation runs, one for each output power configuration of the CC2420 transceiver was carried out for different values of λ. Each simulation run was repeated 30 times with different seeds so that the results could be averaged and assessed by its variance. This configuration is achieved by the following INI file statements:

```
#Defining  the parameter study
**.node[*].nic.mac.txPower = ${power configurations} mW
**.numNodes = ${node densities}*90000
repeat = 30
```

(In the above code, the actual parameters used have been replaced by `power configurations` and `node densities` to save space.)

4.1.4 Line Topology Scenario

In this simulation scenario, nodes were arranged to form a line topology as shown in figure 4.3. A sender node and a sink were placed at opposite ends of the line separated by a fixed distance, D. All the intermediate nodes were separated from the sender in increments of a smaller distance d, such that the number of hops, N, is given by:

$$N = \lceil \frac{D}{d} \rceil \tag{4.2}$$

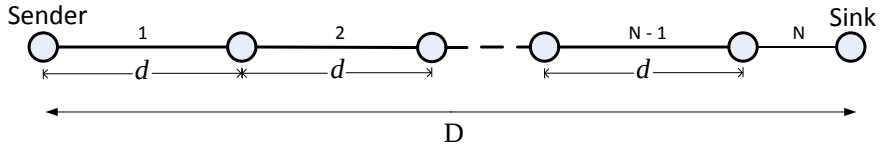

Figure 4.3: Line Topology

The sender then generated 10000 packets that were relayed by all the intermediate nodes to the sink. In each consecutive simulation run the node separation d was increased in steps of 1 meter.

For increasing values of d, the number of hops needed to reach the sink will decrease depending on the ration given in equation 4.2. However, when d reaches a certain point the corresponding PER associated with each link will rise above 0, and retransmission will start to compensate for the loss of packets. It is expected that, as d increases, the total number of transmission will decrease to a minimum around this point, and then start increasing due to retransmissions. Note that, a side effect of using this set up, is that N^{th} hop will consequently be smaller than d to allow fixed positioning of the sender and sink.

Configuration

The configuration of basic node parameters were set in the INI files according to section 4.1.1. Since the topology is changed in each consecutive simulation run, a python script was made to run all simulations and generate the appropriate topology according to the ratio in equation 4.2.

Analytical results

Figure 4.4 shows the expected total number of transmissions versus node range when transmitting to a sink 500 meters away.

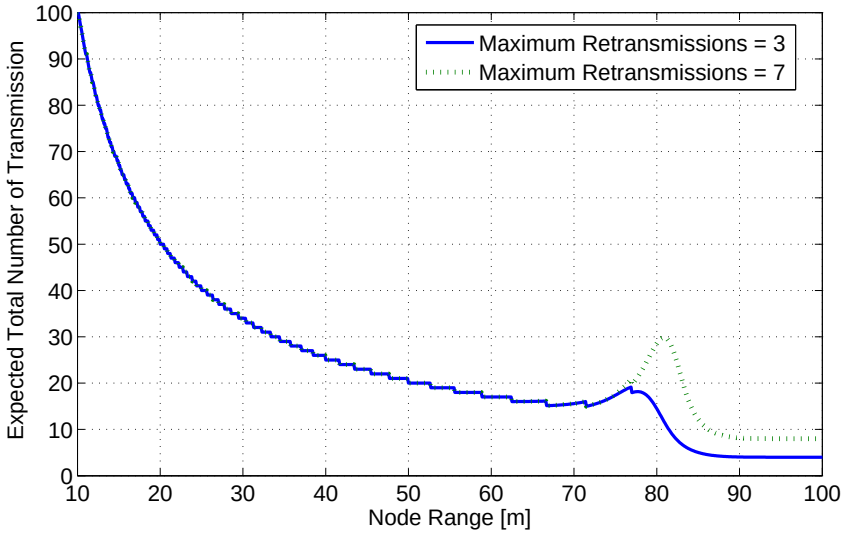

Figure 4.4: Analytical Expected Number of transmissions over a 500 meter distance

4.1.5 Random Rectangle Scenario

This simulation scenario was set up to investigate the effect of the communication range on energy consumption in random networks. The sender and sink were placed at fixed positions in the topology area separated by a distance D. Routes to the sink were then found using RSSIRouting as routing protocol. After all the routes were set up, 1000 packets were sent from the sender to the sink. The simulation was repeated for increasing communication ranges in the interval from 60 to 80 meters. The lower bound of 60 meters was chosen to ensure connectivity in all simulation runs without requiring high node densities. The simulations described in section 4.1.3 showed that nodes separated by a distance higher than about 80 meters will not be able to sustain a 100% end-to-end throughput. For each communication range the simulation was repeated 20 times using different seeds for the random generator. This means that the same 20 topologies were tested for all communication ranges.

Because RSSIRouting was used, the communication range depends on the RSSI threshold parameter that defines the lowest link quality acceptable for routing. To simulate an increasing node range in steps of meters, results from two node simulations were used to set the appropriate RSSI threshold for a given communication range. The RSSI versus distance data from the two node simulations are measured with no interference present. This means that interfering nodes during the route flooding process may influence the actual communication range achieved (interference is not present during data transmissions).

Running the simulations as well as setting the RSSI thresholds was done by scripting. Node separation VS RSSI results were stored in separate vector files and accessed by the script to set the parameters.

Random Topology Generation

The large random network simulations described in section 4.1.3 needed several days to finish using a personal laptop. To be able to terminate experiments in an acceptable time frame, the size of the random network in this scenario was consequently reduced as described by figure 4.5

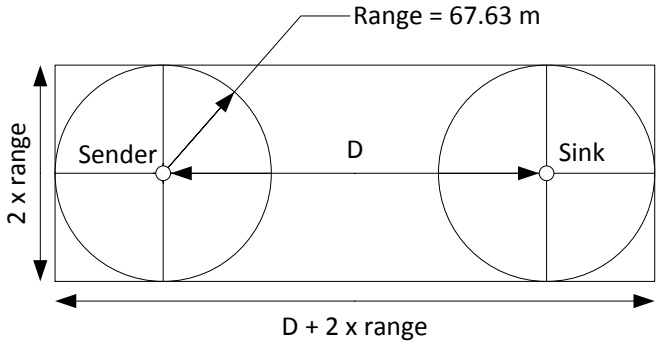

Figure 4.5: Random topology area defined by the distance between the sender and sink and a radius, set to the distance that yields a PER of 1 % (67.63 m for $\alpha = 3.0$ and $P_{tx} = 0dBm$)

4.2 Results

This section presents the results. Since equations for the expected number of transmissions, developed in section 2.5, apply to the line topology scenario, the analytical results will be presented here. The results are briefly analysed and further discussed in section 4.3.

4.2.1 Two Nodes

BER and PER curves

In figure 4.6, simulated PER with packet size of 26 bytes (PSDU = 20 bytes) are shown together with the theoretical BER. The BER curve was obtained using the IEEE 802.15.4 bit error probability formula. The simulated SNR values were averaged over each packet interval, since the simple path loss model was used and the thermal noise is a constant. The PER reaches 1% when the SNR is about 1.5 dB and 100% packet loss is experienced around -2dB.

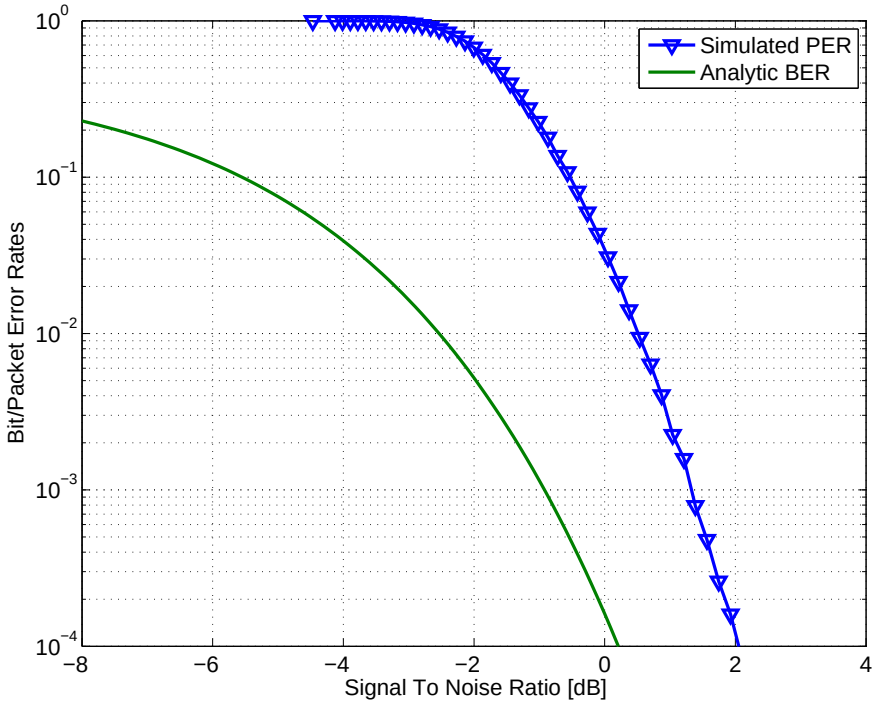

Figure 4.6: Simulated PER with corresponding BER (analytical). The PSDU size is 20 byte.

Simulation and Analytical Results

Figure 4.7 shows the average number of transmissions per packet sent from the sender to the receiver, with maximum allowed retransmissions set to

3 (blue line) and 7 (green with dots). Analytical results using the same parameters as in the simulations are also shown. The simulation results were averaged over 10000 transmissions.

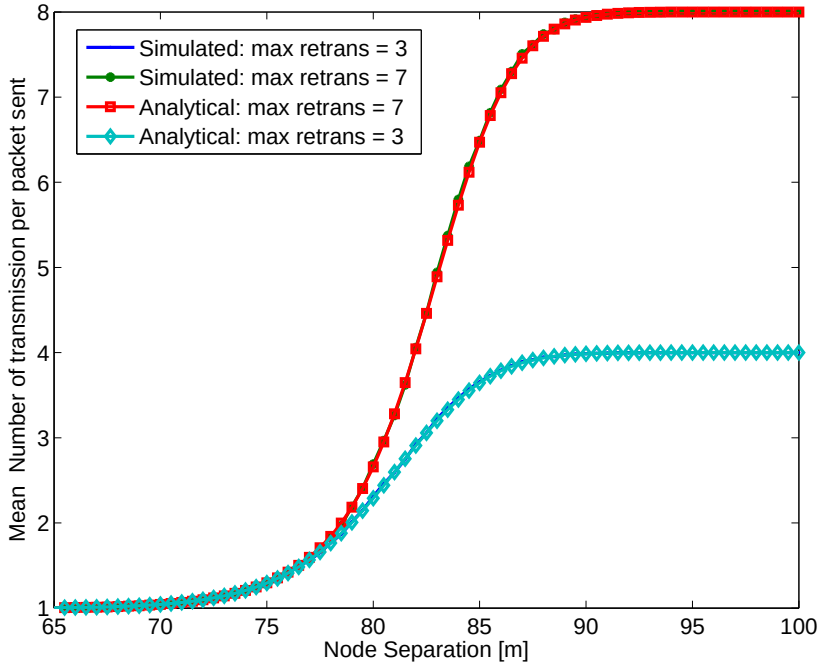

Figure 4.7: Number of transmission on the first hop, simulation VS analytical results

Because the lines coincide so well, the simulation results are difficult to tell a part from the analytical ones. When the node are separated by a distance smaller than approximately 65 meters almost all packets are received successfully, and on average only one transmission is needed per packet. When the node separation reaches about 90 meters, close to all packet fail, and the average number of transmission will consequently be one plus the maximum number of retransmission allowed.

From a point at about 80 meters of node separation, were the curves divide, the PER becomes such that, on average, a higher number of retransmissions are needed to successfully transfer a packet. Beyond this point, 3 retransmission is not enough to make sure that close to all packets are delivered to the receiver. An increasing number of retransmissions from a higher protocol level would be required to guarantee 100% throughput.

PER vs Expected Number of Transmissions

In figure 4.8 the expected number of transmissions on a single hop is plotted against the packet error rate on the link. These results were calculated using the formulas developed in the theory chapter

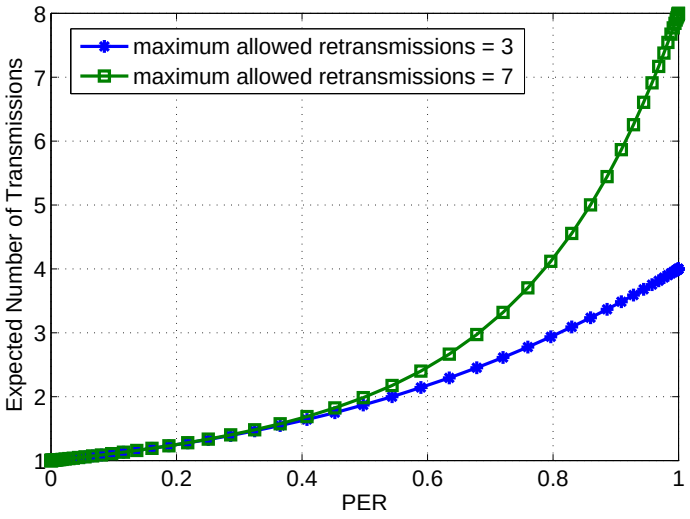

Figure 4.8: PER VS expected number of transmissions (Single Hop)

4.2.2 Line Topology

Analytical VS Simulated Results

In figure 4.9 both analytical and simulation results from the line topology scenario are shown. The sender and sink where separated by a distance equal to 500 meters and the number of retransmissions allowed was set to 100. The average number of transmissions needed, in order successfully relay a packet to the sink, is plotted against node ranges of 10 to 90 meters. The number of transmissions decrease exponentially with increasing node range up to about 70 meters, where a dramatic increase occurs. The steep rise in the number of transmissions is due to retransmission, and shows that even small changes in link quality can have a dramatic effect at high node ranges. The initial decrease in transmissions is due to that fewer hops are needed to reach the sink when using a higher range. This decrease..

The reason why the simulation results show a somewhat better performance is due the border effect mentioned in section 4.1.4. In the analytical approach all hops are considered having the same probability of bit error based on the node range. In the simulations however, the last hop will be of a smaller distance whenever the division of the total distance by the node range yields a remainder.

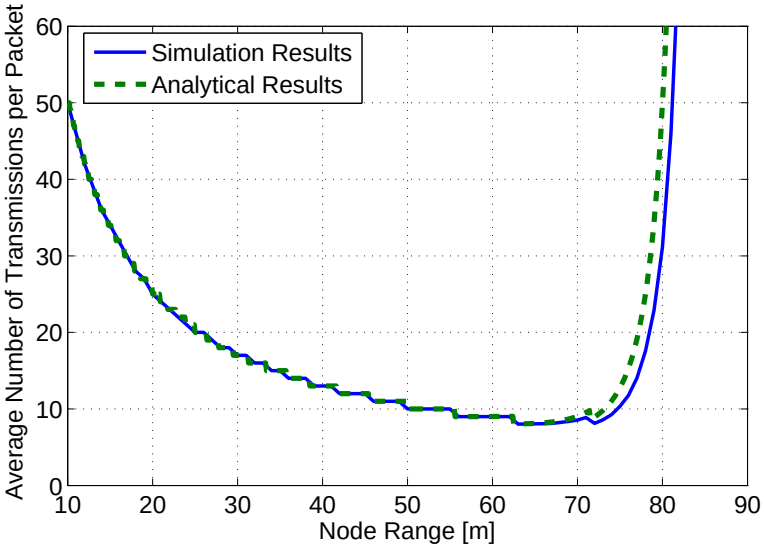

Figure 4.9: The average/expected number of transmissions needed to reach the sink. Both simulated and analytical results are presented as shown in the legend.

End-to-End Throughput

The curves in figure 4.9 does not give a totally correct picture of the number of transmissions needed to reach the sink, since 100 allowed retransmissions is unrealistic. By default the standard sets the to parameter 3, and the maximum value is 7 retransmissions. When the PER is high enough packets will be discarded after a given number of retries, and will have to be retransmitted by the data originator regardless of on which hop the transfer failed (if 100% data reliability is desired).

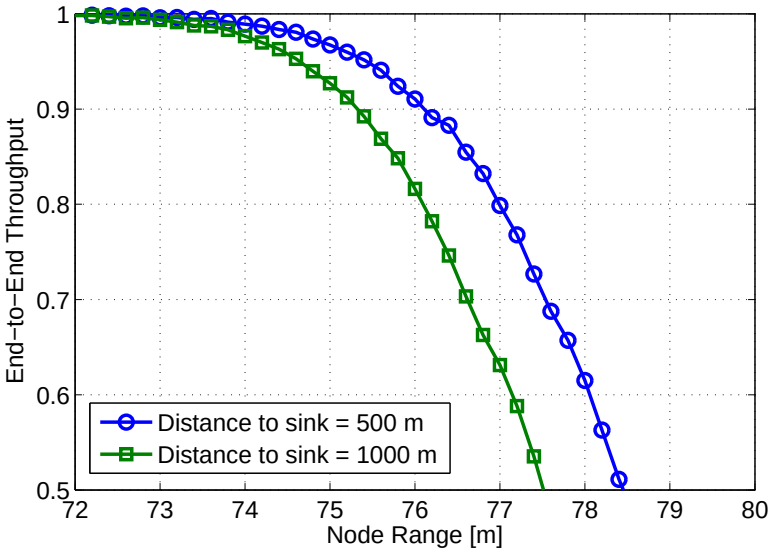

Figure 4.10: Simulated End-to-end throughput versus node range

Figure 4.10 shows how the end-to-end throughput declines in the line topology scenario. The blue line with circle markers shows the simulation results when the distance to the sink was 500 meters, and the green line with square markers the results when the same distance was set to 1000 meters. At 75 meters of node range the number of hops are 7 and 14, for the blue an green line respectively, to reach the sink.

4.2.3 Connectivity in Random Networks

In this section the results from the connectivity simulations described in section 4.1.3 are presented. Note that the nodes model used follows the disk model approach, such that packets traversing an existing route are never dropped.

Connectivity Graphs

The level of connectivity for the different node densities are shown in figures 4.11 (small densities) and 4.12 (large densities). In each figure the blue line with asterisk markers shows connectivity for the lowest densities, the green line with star markers for the middle most densities, and the red line with triangle markers shows the results for the highest node densities. Connectivity is plotted against the different output power configurations as specified by the CC2420 transceiver. None of the node densities represented resulted in connectivity for the lowest output power of -25 dBm. For the highest node desity ($\lambda = 0.005$) however, connectivity was achieved for all output powers above -25dBm. In figure 4.11 it is shown that the node density needed for 100% connectivity at 0 dBm is about 0.0005. The reason why simulations with higher node densities were not performed was that; even with the small densities shown in figure

4.11, the simulation experiment was very time consuming (1-2 days). The simulation experiment for the higher node densities was executed over a 4 days period.

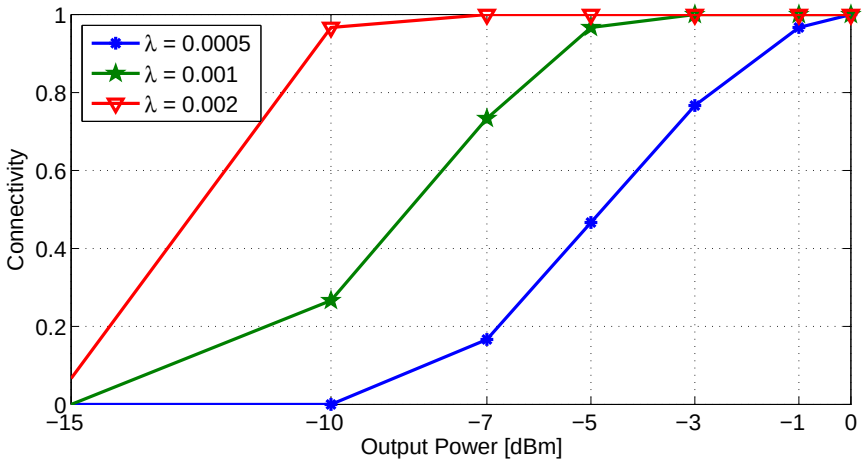

Figure 4.11: Connectivity for Small Node Densities

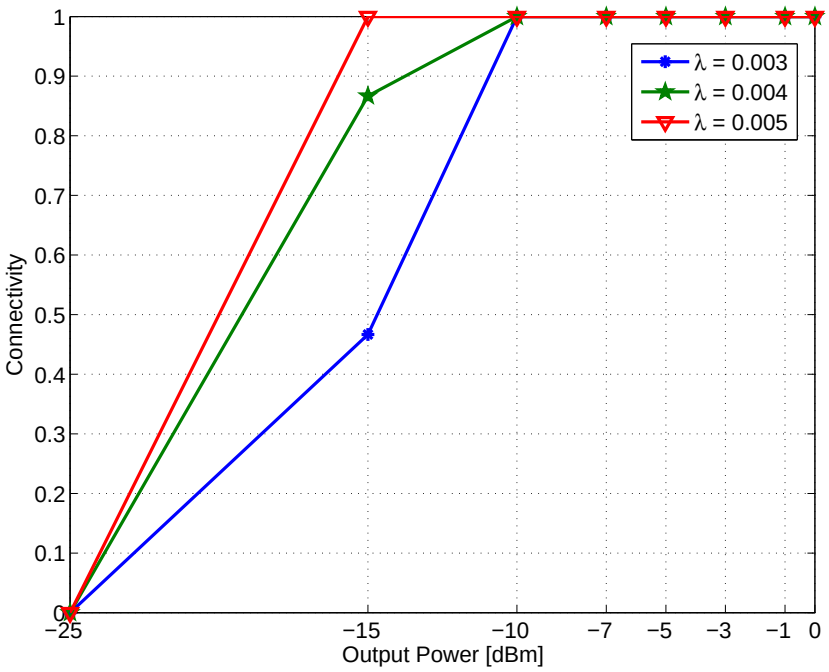

Figure 4.12: Connectivity for Large Node Densities

Number of Hops

The average number of hops needed to reach the sink when $\lambda = 0.005$ is shown in figure 4.13. Results are only shown for the different output power configurations that yielded 100% connectivity at the given node density. The accompanying standard deviation is also shown. The higher output powers result in fewer hops as is expected because a higher range routes can be established through nodes that are closer to the sink. The two highest output power yielded the exact same average number of hops, and the corresponding standard deviations are 0. This is because the lowest number of hops possible for both these configurations also are the same. At -15 dBm the standard deviation is somewhat higher.

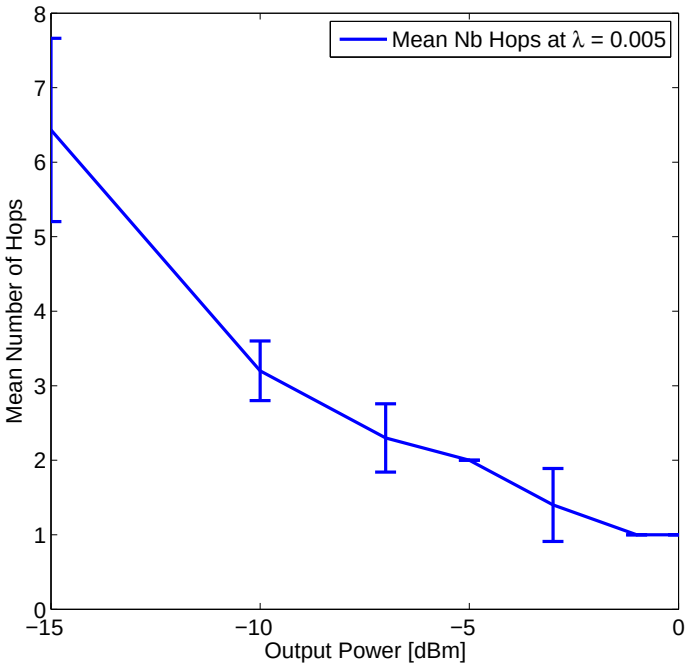

Figure 4.13: Mean Number of hops at $\lambda = 0.005$

In figure 4.14 the average number of hops for the highest 4 output power configurations are plotted against increasing node densities. All curves start with negative slope at $\lambda = 0.001$ that more or less level off for higher node densities. The optimum number of hops at 0 dBm and -1 dBm is 2, since the resulting node ranges are 67.6 and 62.6 respectively. Although the optimum number of hops at -3 dBm is also 2, which yields a range of about 53 meters, 2 hop routes was only achieved about 50% of time from $\lambda = 0.003$ to 0.005.

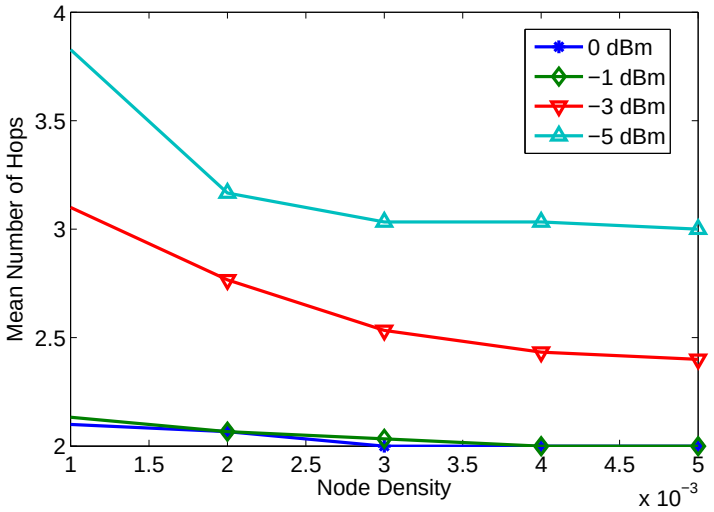

Figure 4.14: Mean Number of hops at $\lambda = 0.005$

4.2.4 Random Rectangle Scenario

Though the line topology provides useful information on how retransmission effect the energy consumption in the extreme case, a perfect line topology is usually not realistic. This subsection presents the results from the random topology scenario as described in section 4.1.5.

Energy Consumption

The total average energy consumed by the network in order to successfully relay a packet to the sink 300 meters away is shown in figure 4.15. The energy consumption is plotted against communication ranges from 60 to 80 meters. Energy consumed by transmission and reception of both data and ACK packets are taken into account. Three different values for the maximum number of retransmissions allowed parameter were simulated as depicted by the legend. Though 7 (green line with plus markers) is the maximum value for the IEEE 802.15.4 standard, simulations with the same parameter set to 100 (red line with diamonds) was included to illustrate that even more retransmissions are needed to achieve 100 % end-to-end throughput. At a communication range of 60 meters a higher number of hops (about 6.3 on average) are needed to reach the sink, but all packets are transferred without error and no retransmissions are needed. Expanding the range yields a fewer number of hops, and the energy benefit reaches a maximum at about 71-72 meters.

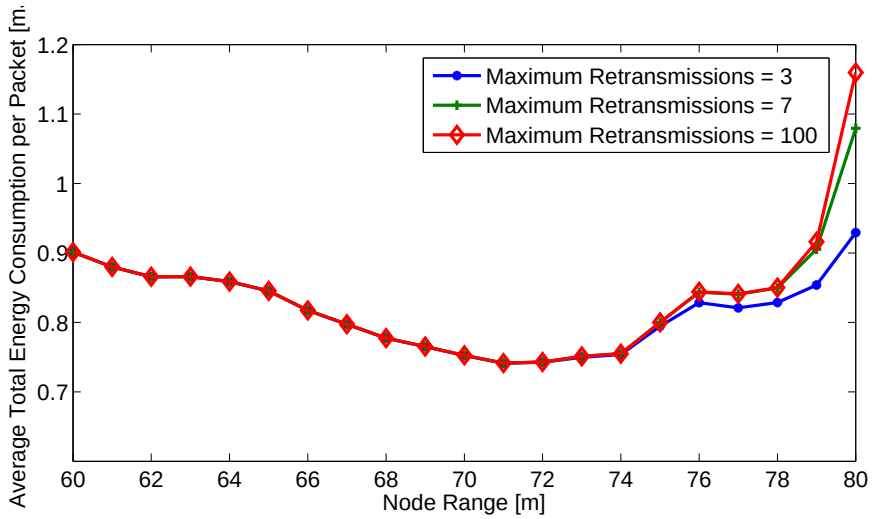

Figure 4.15: Per packet energy consumption in a 100 node random network topology. The distance from the sender to the sink is 300 m.

Figure 4.16 shows how retransmissions influence the overall energy consumption due to packet transmissions alone. The blue line with square markers shows the total energy consumption spent on data packet transmissions while the green line with diamond markers is the energy consumed by retransmissions. The yellow line with plus sign markers is the difference between the two other curves and gives the energy consumed if no retransmission were needed.

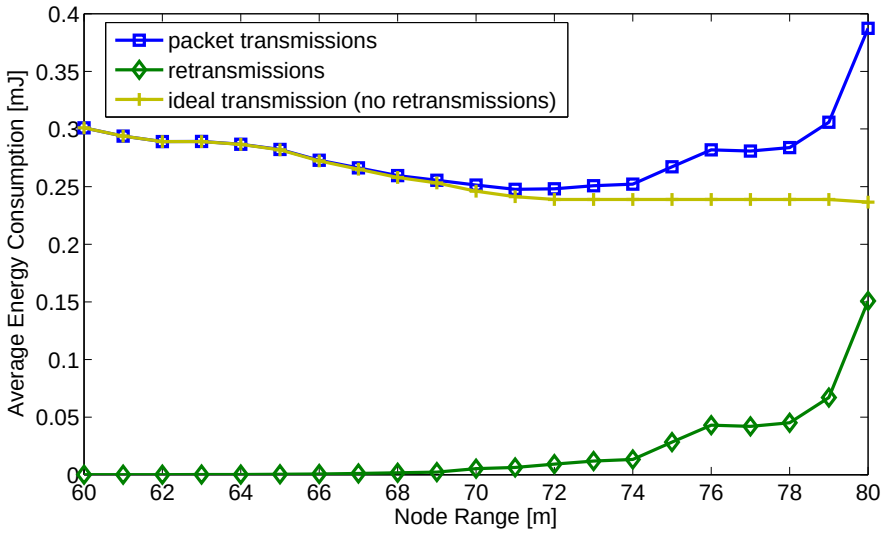

Figure 4.16: Per packet sent energy consumption due to transmissions. Packets are sent to a sink 300 meters by multi-hop communication in a random topology network with 100 nodes.

Number of Hops versus Communication Range

Figure 4.17 shows the mean number of hops needed to reach the sink plotted against node range. The standard error is shown as vertical bars at each point of sample means. The mean number of hops decreases with approximately the same rate throughout the graph, but it levels off in the intervals between 62-63 and 72-79 meters of node range. For the whole latter interval the standard error is '0', indicating that all simulation runs resulted in 5 hop routes to the sink. At 80 meters the mean number of hops drops below 5. Theoretically, the probability of a 4 hop route should be present with communication ranges above 75 meters when the total distance is 300. Depending on the node density the probability of a 5 hop route is expected to reach a maximum relatively close to and above 75 meter. Similarly the probability of a 6 hop route is expected to reach a maximum close to an above 60 meters. The graph indicate that no benefit of fewer hops is gained by increasing the communication range within the intervals described.

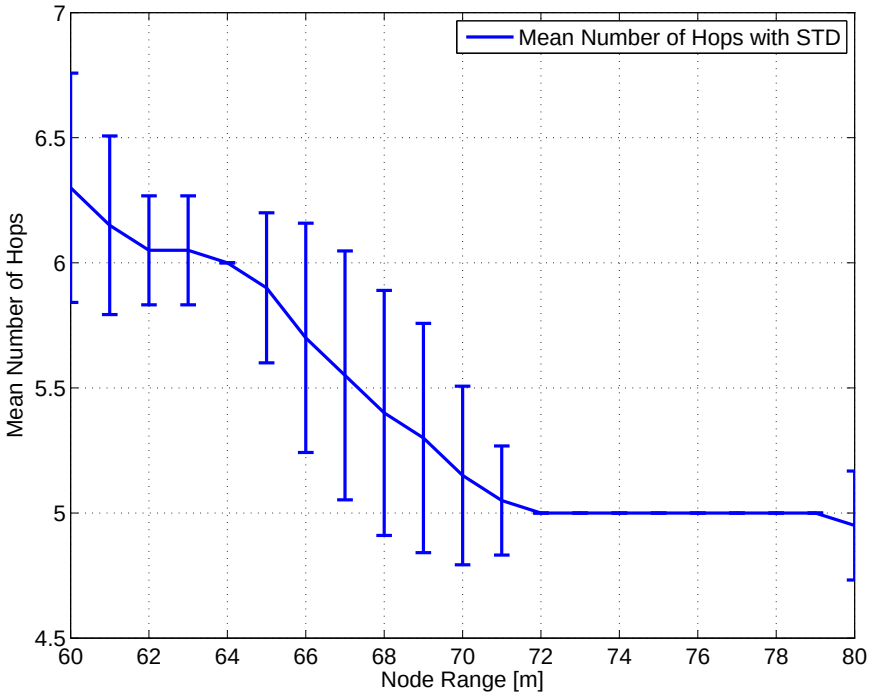

Figure 4.17: Mean number of hops to reach the sink 300 meters away, in a random topology network.

4.3 Discussion

From the results it is clear that, by increasing the communication range up to a certain point an energy benefit can be gained by routing the packets over a fewer number of hops. A benefit is only gained if the increase in node range yields a fewer number of hops, which depends on both the node density and the distance to the sink. However, if the range is increased beyond a certain point, retransmissions will quickly dominate the number of transmissions needed to reach the sink. Occasional packet loss can be dealt with using the up to 7 maximum allowed retransmission as prescribed by the IEEE 802.15.4 standard. When 100% data reliability is required, packets must be retransmitted in the next available time slot from the data originator, if a transmission fails after maximum allowed retries. This leads to additional energy consumption per packet, as well as causing excess delay which could have devastating effects on real time applications. The cost of a transmission failure is clearly greater when the packet fails on the last hop to the sink, since a retransmission from the originator implies that the packet must follow all the same hops once again. Thus a higher link quality should be required for nodes close to the sink. Additionally, more traffic is likely to pass through these nodes, further emphasising the need for high quality links.

Using RSSI to measure the link quality is a common simplification in simulations [27]. RSSI could be used directly to indicate the quality of the link, but has the disadvantage that interference imposed on the signal may increase the measured LQI, when the true link quality is actually reduced. I is also pointed out in [25] that other measurements for LQI provides a higher statistical relation with the true PER. As recommended by [6] the LQI should be found empirically through PER measurements.

Hidden terminals may cause interference when route floods are received. In the extreme case, when several floods arrive at a node simultaneously, with signal levels in the same order of magnitude, the receiving node will not be able to receive any of them. However, if the signal level of a single route flood is With RSSI as metric, interference imposed on a node during the reception of a route flood will falsely suggest a high quality link, when in fact the case is the opposite.

Chapter 5

Conclusion

In this thesis it is shown that a minimum point for energy consumption in TDMA base, random topology WSN can be found. However, if the quality of the link is slightly degraded from this point, the energy consumption may increase drastically due to retransmission. Also, the effect of a packet failure is more devastating on links closer to the sink. Consequently the nodes close to the sink should have higher requirement for link quality. Additionally more traffic is likely to pass through these nodes. An LQI based on PER measurements should be used to ensure that links chosen reflect the true link quality.

Investigating large random topology networks through simulation, is highly limited by the scalability of the network simulator used. A lot of effort was made to remove model components, not relevant for the experiments, to enhance the performance of the simulation model. Complexity should only be added to the model where it has an enhancing impact on the quality of the results.

Bibliography

[1] http://www.omnetpp.org/.

[2] http://inet.omnetpp.org/.

[3] http://castalia.research.nicta.com.au/index.php/en/.

[4] http://mixim.sourceforge.net/.

[5] I.F. Akyildiz et al. 'A Survey on Sensor Networks'. In: *Communications Magazine, IEEE* 40.8 (2002), pp. 102–114.

[6] *CC2420 Data Sheet*. Oslo, Norway: Chipcon Products from Texas Instruments.

[7] Chee-Yee Chong and S.P. Kumar. 'Sensor Networks: Evolution, Opportunities, and Challenges'. In: *Proceedings of the IEEE* 91.8 (Aug. 2003), pp. 1247–1256.

[8] E. Egea-Lopez et al. 'Simulation scalability issues in wireless sensor networks'. In: *Communications Magazine, IEEE* 44.7 (2006), pp. 64–73.

[9] Peder J. Emstad et al. *Dependability and performance in information and communication systems - Fundamentals*. tapir, 2011.

[10] C.C. Enz, N. Scolari and U. Yodprasit. 'Ultra low-power radio design for wireless sensor networks'. In: *Radio-Frequency Integration Technology: Integrated Circuits for Wideband Communication and Wireless Sensor Networks, 2005. Proceedings. 2005 IEEE International Workshop on*. 2005, pp. 1–17.

[11] Yu Feng et al. 'A Localization Algorithm for WSN Based on Characteristics of Power Attenuation'. In: *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08. 4th International Conference on*. 2008, pp. 1–5.

[12] M. Goyal et al. 'Evaluating the Impact of Signal to Noise Ratio on IEEE 802.15.4 PHY-Level Packet Loss Rate'. In: *Network-Based Information Systems (NBiS), 2010 13th International Conference on*. 2010, pp. 279–284.

[13] M. Haenggi. 'Twelve reasons not to route over many short hops'. In: *Vehicular Technology Conference, 2004. VTC2004-Fall. 2004 IEEE 60th*. Vol. 5. 2004, 3130–3134 Vol. 5.

[14] M. Haenggi and D. Puccinelli. 'Routing in ad hoc networks: a case for long hops'. In: *Communications Magazine, IEEE* 43.10 (2005), pp. 93–101.

[15] Anne-Lena Kampen et al. 'Energy Reduction in Wireless Sensor Networks by Switching Nodes to Sleep During Packet Forwarding'. In: *SENSORCOMM 2012 , The Sixth International Conference on Sensor Technologies and Applications*. Rome, Italy, Aug. 2012, pp. 189–195.

[16] Holger Karl and Andreas Willing. *Protocols and Architectures for Wireless Sensor Networks*. WILEY, 2007.

[17] Andreas Köpke et al. 'Simulating Wireless and Mobile Networks in OMNeT++ – The MiXiM Vision'. In: *OMNeT++ 2008: Proceedings of the 1st International Workshop on OMNeT++ (hosted by SIMUTools 2008)*. Marseille, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[18] Dust Networks. *Dust Networks Applications*. http://www.linear.com/designtools/wireless_sensor_apps.php. Webside med Dust Networks applikasjonsomrÃ¥der. 2012.

[19] *OMNeT++ User Manual*.

[20] *PART: 15.4 Low-Rate Wireless Personal Area Networks*. IEEE Computer Society, 2011.

[21] J. Rousselot et al. 'Accurate Timeliness Simulations for Real-Time Wireless Sensor Networks'. In: *Computer Modeling and Simulation, 2009. EMS '09. Third UKSim European Symposium on*. 2009, pp. 476–481.

[22] Bernard Sklar. *Digital Communications: Fundamentals and Applications*. **Review:** *IEEE Communications*, Vol. 27, No. 8, August 1989. Prentice Hall, 2001.

[23] Kannan Srinivasan et al. 'An empirical study of low-power wireless'. In: *ACM Trans. Sen. Netw.* 6.2 (Mar. 2010), 16:1–16:49. ISSN: 1550-4859.

[24] Andriy Stetsko et al. 'On the credibility of wireless sensor network simulations: evaluation of intrusion detection system'. In: *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*. 2012.

[25] Lei Tang et al. 'Channel Characterization and Link Quality Assessment of IEEE 802.15.4-Compliant Radio for Factory Environments'. In: *Industrial Informatics, IEEE Transactions on* 3.2 (2007), pp. 99–110.

[26] András Varga. 'The OMNeT++ Discrete Event Simulation System'. In: *Proceedings of the European Simulation Multiconference (ESM'2001)* (June 2001).

[27] Klaus Wehrle, Mesut Günes and James Gross, eds. *Modeling and Tools for Network Simulation*. Springer, 2010. ISBN: 978-3-642-12330-6. URL: http://dx.doi.org/10.1007/978-3-642-12331-3.

[28] A. Willig, K. Matheus and A. Wolisz. 'Wireless Technology in Industrial Networks'. In: *Proceedings of the IEEE* 93.6 (2005), pp. 1130–1151.

Appendix A

Derivation of formulas

A.1 Expected Number of transmissions

A.1.1 Single hop

When transmitting packets on a link with a probability of packet loss q, the expected number of transmissions per packet is given by:

$$E\{X_{tx}\} = 1 \cdot q^0(1-q) + 2 \cdot q^1(1-q) + 3 \cdot q^2(1-q) + ... \quad (A.1)$$

$$... + m \cdot q^{m-1} \cdot (1-q) + m \cdot q^m \quad (A.2)$$

or;

$$E\{X_{tx}\} = \sum_{k=1}^{m} \left(k \cdot q^{k-1} \cdot (1-q) \right) + m \cdot q^m \quad (A.3)$$

where m is the maximum number of transmissions, and the maximum number of retransmissions is $(m-1)$.

In equation A.3, $(1-q)$ is a constant, and can be factored out. By denoting $S = \sum_{k=1}^{m} k \cdot q^{k-1}$, the equation can be re-written as:

$$E\{X_{tx}\} = S \cdot (1-q) + m \cdot q^m \quad (A.4)$$

The sum,S, can be expressed as:

$$S = 1 \cdot q^0 + 2 \cdot q^1 + ... + m \cdot q^{m-1} \quad (A.5)$$

Subtracting $S \cdot q$ from S yields:

$$S(1-q) = 1 \cdot q^0 + 2 \cdot q^1 + ... + m \cdot q^{m-1} \quad (A.6)$$

$$- (1 \cdot q^1 + 2 \cdot q^2 + ... + n \cdot q^m) \quad (A.7)$$

$$= (q^0 + q^1 + q^2 + ... + q^{m-1}) - m \cdot q^m \quad (A.8)$$

where $(q^0 + q^1 + q^2 + ... + q^{m-1})$ is a geometric series, and can be expressed as:

$$\frac{1 - q^{(m-1)+1}}{1 - q} = \frac{1 - q^m}{1 - q} \tag{A.9}$$

S then simplifies to:

$$S = \frac{1 - q^m}{(1 - q)^2} - \frac{m \cdot q^m}{(1 - q)} \tag{A.10}$$

Replacing S in equation A.4 with A.10 yields:

$$E\{X_{tx}\} = \left(\frac{1 - q^m}{(1 - q)^2} - \frac{m \cdot q^m}{(1 - q)}\right)(1 - q) + m \cdot q^m = \frac{1 - q^m}{1 - q} \tag{A.11}$$

Appendix B

Python Scripts

B.1 Random Topology Generation

```
#!/usr/bin/env python
import sys, math, re, os, string
from numpy import random

#This script creates a random rectangular topology and ↩
    outputs an file (ini-syntax) compatible with mixim
#The rectangle is centered in the playground area
#Nodes '0' and '1' are placed at opposite ends of the ↩
    rectang (both with y-coordinate = playGroundY/2)

#Parameters to fill inn
playGorundX = 600
playGorundY = 400
height = 50
lenght = 100
numNodes = 50

generatedNodes = 2
centerX = playGorundX/2
centerY = playGorundY/2

#Opening File for Writing
resHandle = file('topologyTest','a')
resHandle.write("#This topology file yields a ↩
    rectangular topology centered in a "+ str(↩
    playGorundX) + "by" +str(playGorundY) +" playgound ↩
    area with: Height = " + str(height) + " and " + "↩
    Length = " + str(lenght) +"\n")
resHandle.write("#There are " + str(numNodes) + " node ↩
    entries " + "]\n")

#
resHandle.write('**.numNodes =' + str(numNodes) + '\n')
#Positioning sink and sender node
resHandle.write('**.node[0].mobility.initialX = ' + str↩
```

```
    (centerX-lenght/2) + 'm\n')
resHandle.write('**.node[1].mobility.initialX = ' + str↩
    (centerX+lenght/2) + 'm\n')
resHandle.write('**.node[0].mobility.initialY = ' + str↩
    (centerY) + 'm\n')
resHandle.write('**.node[1].mobility.initialY = ' + str↩
    (centerY) + 'm\n')
while generatedNodes < numNodes:
  initialX = random.uniform(centerX-lenght/2,centerX+↩
     lenght/2)
  initialY = random.uniform(centerY-height/2,centerY+↩
     height/2)
  print str(initialX) + ", " + str(initialY)
  resHandle.write('**.node[' + str(generatedNodes)+'].↩
     mobility.initialX = ' + str(initialX) + 'm\n')
  resHandle.write('**.node[' + str(generatedNodes)+'].↩
     mobility.initialY = ' + str(initialY) + 'm\n')
  generatedNodes += 1
```

Appendix C

Matlab Scripts

Appendix D

OMNeT++ Coding

D.1 The MittNettverk Module

```
package minruting;

import org.mixim.base.modules.BaseNetwork;
//Nettverk
network MittNettverk extends BaseNetwork
{
    parameters:
        int numNodes;
        double placeHolder = default(0.0);
        @statistic[totRetrans](title="totRetrans"; ↩
            source="retransSignal"; record=vector?,↩
            stats?; interpolationmode=none);
        @statistic[totTxFrames](title="totTxFrames"; ↩
            source="txFramesSignal"; record=vector?,↩
            stats?; interpolationmode=none);
      @statistic[totRxFrames](title="totRxFrames"; ↩
          source="rxFramesSignal"; record=vector?,stats↩
          ?; interpolationmode=none);
      @statistic[totReceivedAcks](title="↩
          totReceivedAcks"; source="receivedAckSigal"; ↩
          record=vector?,stats?; interpolationmode=none↩
          );
      @statistic[totTxAcks](title="totTxAcks"; source="↩
          txAckSignal"; record=vector?,stats?; ↩
          interpolationmode=none);

    submodules:
        node[numNodes]: SensorNode;
}
```

D.2 The implementation of SensorNode

Figure D.1 shows the NED inheritance diagram for the `Host802154_2400MHz` module. The `WirelessNodeNetwl` module defines a basic node implementation with NIC, application and network layer modules. The trans-

port layer module as well as connections between the different protocol layer submodules are defined in `WirelessNodePlusTran`, and the battery module is added in `WirelessNodeBatteryPlusTran`.Implementing `SensorNode` without the transport layer and battery modules can be done by extending another MiXiM module called `WirelessNode`. This module defines a wireless node with the exact same submodules as in `WirelessNodeNetwl`, but provides the appropriate connections to go with it. The `Host802154_2400MHz` module implements the `Nic802154_TI_CC2420` as NIC module. Because this NIC was designed to work with the battery module it extends `WirelessNicBattery` which provides the necessary parameters (e.g. rxCurrent, txCurrent etc.). To make the `Nic802154_TI_CC2420` module work together with `SensorNode`, which does not provide a battery module, it must extend a more general module called `WirelessNic`. The NED inheritance diagrams for both `SensorNode` and the modified version of the `Nic802154_TICC2420` module are shown in figure D.2

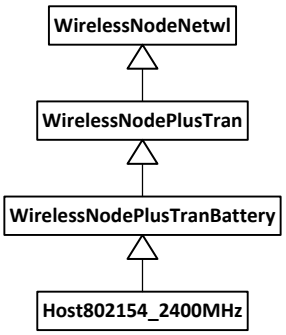

Figure D.1: Host802154_2400MHz NED inheritance diagram

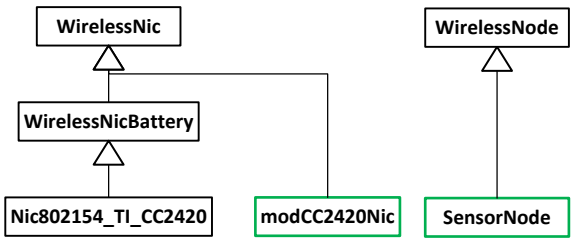

Figure D.2: SensorNode and NIC NED Inheritance diagram

D.2.1 SensorNode NED definition

```
//This module models a sensornode with a CC2420 radio ↩
    tranciever

package minruting;

import org.mixim.modules.node.WirelessNode;
import org.mixim.modules.nic.WirelessNic;

module SensorNode extends WirelessNode
{
    parameters:
        applicationType = default("SensorApplLayer");
        nicType = default("modCC2420Nic");  //type of ↩
            used nic
        arpType = default("org.mixim.modules.netw.↩
            ArpHost");
        networkType = default("RSSIruting");

        @display("bgb=210,451;i=abstract/router;is=vs")↩
            ;
        @statistic[retransmisjoner](title="↩
            retransmisjoner"; source="retransSignal"; ↩
            record=vector?,stats?; interpolationmode=↩
            none);
        @statistic[macRssiSignal](title="macRssiSignal"↩
            ; source="macRssiSignal"; record=stats?; ↩
            interpolationmode=none);
        @statistic[macSnrSignal](title="macSnrSignal"; ↩
            source="macSnrSignal"; record=stats?; ↩
            interpolationmode=none);
        @statistic[macBerSignal](title="macBerSignal"; ↩
            source="macBerSignal"; record=stats?; ↩
            interpolationmode=none);
        @statistic[packetReceived](title="↩
            packetReceived"; source="packetReceived"; ↩
            record=stats?; interpolationmode=none);
        @statistic[dataPacketSent](title="↩
            dataPacketSent"; source="dataPacketSent"; ↩
            record=stats?; interpolationmode=none);
        @statistic[TxFrames](title="TxFrames"; source="↩
            txFramesSignal"; record=vector?,stats?; ↩
            interpolationmode=none);
        @statistic[RxFrames](title="RxFrames"; source="↩
            rxFramesSignal"; record=vector?,stats?; ↩
            interpolationmode=none);
}
```

D.2.2 modCC2420Nic

```
//This is the module definition for MAC an PHY layer
module modCC2420Nic extends WirelessNic
{
    macType = default("CSMA802154");
```

```
    phy.decider = default(xmldoc("↩
        Nic802154_TI_CC2420_Decider.xml"));
    phy.headerLength = 48 bit; // ieee 802.15.4
    phy.thermalNoise = default(-110 dBm);
    // From TI CC1100 datasheet rev. C
    phy.timeSleepToRX = 0.001792 s;
    phy.timeSleepToTX = 0.001792 s;
    phy.timeRXToTX = 0.000192 s;
    phy.timeTXToRX = 0.000192 s;
    phy.timeRXToSleep = 0 s;
    phy.timeTXToSleep = 0 s;
    mac.rxSetupTime = 0.001792 s;
}
```

D.3 Network Layer Implementations

D.3.1 Closest Routing

ClosestRouting.ned

```
package minruting;
import org.mixim.base.modules.BaseNetwLayer;

simple ClosestRouting extends BaseNetwLayer
{
    parameters:
        @class(ClosestRouting);
}
```

ClosestRouting.h

```
#ifndef __MINRUTING_CLOSESTROUTING_H_
#define __MINRUTING_CLOSESTROUTING_H_

#include <omnetpp.h>
#include "MiXiMDefs.h"
#include "BaseNetwLayer.h"
#include "SimpleAddress.h"

class SimTracer;
class WiseRoutePkt;

class MIXIM_API ClosestRouting : public BaseNetwLayer
{
    private:
        simsignal_t packetReceived;
        simsignal_t dataPacketSent;

    public:
        virtual void initialize(int);
    protected:
```

```
        virtual void handleUpperMsg(cMessage* msg);

        virtual void handleLowerMsg(cMessage* msg);

        virtual void handleSelfMsg(cMessage* msg);

        virtual void handleLowerControl(cMessage* msg);

        cMessage* decapsMsg(WiseRoutePkt *msg);
};

#endif
```

ClosestRouting.cc

```
#include "ClosestRouting.h"
#include "NetwControlInfo.h"
#include "MacToNetwControlInfo.h"
#include "ArpInterface.h"
#include "FindModule.h"
#include "WiseRoutePkt_m.h"
#include "SimTracer.h"
#include "ChannelAccess.h"

using std::make_pair;
Define_Module(ClosestRouting);

void ClosestRouting::initialize(int stage)
{
    BaseNetwLayer::initialize(stage);
    packetReceived = registerSignal("packetReceived");
    dataPacketSent = registerSignal("dataPacketSent");
}

void ClosestRouting::handleUpperMsg(cMessage* msg)
{
    emit(dataPacketSent,1);
    LAddress::L3Type finalDestAddr;
    LAddress::L3Type nextHopAddr;
    LAddress::L2Type nextHopMacAddr;
    WiseRoutePkt*    pkt    = new WiseRoutePkt(msg->↩
       getName());
    cObject*         cInfo = msg->removeControlInfo();

    pkt->setByteLength(headerLength);

    if ( cInfo == NULL ) {
        EV << "WiseRoute warning: Application layer did↩
            not specifiy a destination L3 address\n"
               << "\tusing broadcast address instead\n↩
                  ";
        finalDestAddr = LAddress::L3BROADCAST;
```

```
    }
    else {
        EV <<"WiseRoute: CInfo removed, netw addr="<< ↩
            NetwControlInfo::getAddressFromControlInfo(↩
             cInfo ) <<endl;
        finalDestAddr = NetwControlInfo::↩
            getAddressFromControlInfo( cInfo );
        delete cInfo;
    }

    pkt->setFinalDestAddr(finalDestAddr);
    pkt->setInitialSrcAddr(myNetwAddr);
    pkt->setSrcAddr(myNetwAddr);
    pkt->setNbHops(0);

    if (LAddress::isL3Broadcast(finalDestAddr))
        nextHopAddr = LAddress::L3BROADCAST;
    else
        nextHopAddr = myNetwAddr-1;

    pkt->setDestAddr(nextHopAddr);

    if (LAddress::isL3Broadcast(nextHopAddr)) {
        nextHopMacAddr = LAddress::L2BROADCAST;
    }
    else {
        nextHopMacAddr = arp->getMacAddr(nextHopAddr);
    }

    setDownControlInfo(pkt, nextHopMacAddr);
    assert(static_cast<cPacket*>(msg));
    pkt->encapsulate(static_cast<cPacket*>(msg));
    sendDown(pkt);
}


void ClosestRouting::handleLowerMsg(cMessage* msg)
{
    emit(packetReceived,1);
    WiseRoutePkt*           netwMsg        = ↩
        check_and_cast<WiseRoutePkt*>(msg);
    const LAddress::L3Type& finalDestAddr  = netwMsg->↩
        getFinalDestAddr();
    netwMsg->setSrcAddr(myNetwAddr);
    const cObject* pCtrlInfo = NULL;
    pCtrlInfo = netwMsg->removeControlInfo();

    if (finalDestAddr == myNetwAddr){
        sendUp(decapsMsg(netwMsg));
    }
    else{
        LAddress::L3Type nextHopAddr = myNetwAddr-1;
        netwMsg->setSrcAddr(myNetwAddr);
        netwMsg->setDestAddr(nextHopAddr);
```

```
            netwMsg->setNbHops(netwMsg->getNbHops()+1);
            setDownControlInfo(netwMsg, arp->getMacAddr(↩
                nextHopAddr));

            sendDown(netwMsg);
        }
        if (pCtrlInfo != NULL)
            delete pCtrlInfo;
    }

    void ClosestRouting::handleSelfMsg(cMessage* msg)
    {
        EV <<"We Received A self message, deleting it..."<<↩
            endl;
        delete msg;
    }

    void ClosestRouting::handleLowerControl(cMessage* msg)
    {
        delete msg;
    }

    cMessage* ClosestRouting::decapsMsg(WiseRoutePkt *msg)
    {
        cMessage *m = msg->decapsulate();
        setUpControlInfo(m, msg->getSrcAddr());
        // delete the netw packet
        delete msg;
        return m;
    }
```

D.3.2 RSSIRouting